

Department of Computer
and Information Science
PD. Dr. Torsten Grust
Prof. Dr. Marc H. Scholl

An XQuery to SQL Compiler

Bachelor Thesis

Brendan Briody
01/461643
Hohenhausgasse 14
78462 Konstanz
mail@brendan.de

Konstanz, 23rd November 2004

Abstract

This thesis presents the implementation of an XQuery to SQL compiler as a back-end solution to the Pathfinder project. The compiled SQL queries can be executed to query encoded XML documents on Relational Database Management Systems (RDBMS). These database systems are widely used in data storage and retrieval. They are efficient on indexed scans and are known to cope well with large amounts of data. Therefore, RDBMS can become useful as a backend XML storage system for XQuery.

The compilation of XQuery by Pathfinder creates abstract syntax trees which are used by the implementation to generate SQL code. For this code generation the *twig* compiler abilities used by the Pathfinder project are shown to be very useful. Performance tests of compiled SQL code on different sized documents show promising results. However further optimisations concerning the XQuery to SQL compiler and the availability of special built in functions on RDBMS towards XQuery needs are certainly desirable.

Contents

1	Introduction	1
1.1	Aims	3
1.2	Structure of this Thesis	4
2	XQuery and Relations	6
2.1	An Introduction to XQuery	6
2.2	Iterative and Set Oriented Queries	9
3	XQuery to SQL Translation Rules	15
3.1	The Result and Document Schemes	15
3.2	Inference Rules	17
3.2.1	Simple Rules	19
3.2.2	Complex Rules	21
4	XQuery to SQL Implementation	33
4.1	SQL Tree Expressions	33
4.2	XQuery Pattern Matching	39
4.2.1	An Introduction to the Twig Compiler	39
4.2.2	Twig Implementation	42
5	Performance Tests	61
5.1	Testing Procedures	61
5.2	Occurring Problems	63
5.3	Result Tests	63
5.4	Time Measurements	67
6	Discussion	70
6.1	Relational Encoding	70
6.2	XQuery to SQL Refinements	71
6.3	Future Development on RDBMS	73

List of Figures

1.1	High-Level Taxonomy by Krishnamurthy	3
1.2	The Pathfinder Project with the XQuery to SQL Compiler	5
2.1	A fraction of the "books.xml" document	8
2.2	Lifted encoding of Query 2.1 and the result as a flattened table . . .	10
2.3	FLWOR scopes and resulting scope tree	11
2.4	Scopes on Query 2.1	11
3.1	Simple relational encodings of items	15
3.2	The full relational encodings of XML documents / fragments . . .	16
3.3	XQuery core expressions	17
3.4	General top-down to bottom-up compilation	18
3.5	An XML fragment	22
3.6	The tree representation of the XML fragment in Figure 3.5	24
3.7	Numbering without OLAP	26
3.8	Map relation without OLAP	26
3.9	DENSE_RANK() on the map relation	26
4.1	The structure type for SQL nodes	34
4.2	The storage structure of SQL nodes with wired child nodes	36
4.3	Resulting SQL tree	38
4.4	The <i>twig</i> rule notation	39
4.5	Twig rules for a plus expression	40
4.6	A tree pattern and its corresponding label access	41
4.7	The C structure types for XQuery and SQL nodes	43
4.8	The <i>CONST</i> inference rule in <i>twig</i>	44
4.9	The <i>SEQ</i> inference rule in the <i>twig</i> rule	45
4.10	Action code for a variable occurrence.	46
4.11	The LET tree pattern with <i>twig</i> access pointers	47
4.12	LET binding of a variable together with environment handling. . . .	48
5.1	A fraction of a document from XMLgen	61

5.2	Compiled SQL code of the sequence (10,(20,30)	64
5.3	XML documents tested by Query 5.5	69

Code and Table Index

1.1	An excerpt of relevant techniques by Krishnamurthy	3
2.1	A simple nested FLWOR	10
2.2	The sequence of $\$x(a)$ and $\$x$ in scope S_0 (b)	12
2.3	$\$y$ (a), the map (b) and $\$y$ in $S_{0.1}$ (c)	12
2.4	$\$x$ in S_0 (a), the map for $\$x$ to $S_{0.1}$ (b) and the result for $\$x$ (c) .	13
2.5	Fully loop lifted $\$x$ (a) and $\$y$ (b) in $S_{0.1}$ and the addition result (c)	13
2.6	Back mapping of the result up to scope S	13
2.7	Query 2.1 with a further binding to $\$t$	14
3.1	The sequence (10,20,30) with calculated positions	20
3.2	XML fragment of Figure 3.5 as a relation	23
3.3	The duplicates created by multiple joins of Table 3.2	23
3.4	The result from the path query	31
3.5	The new root \mathbf{y}	31
3.6	The result schema of the new root \mathbf{y}	31
3.7	The original document with frag n and the new document frag $n+1$	32
5.1	The result of the sequence and the loop table	63
5.2	The result of the nested FLWOR	65
5.3	The element result	66
5.4	The new document frag 2	66
5.5	Tested FLWOR query	67
5.6	Result from B.document	67
5.7	New document fragments from B.document	68
6.1	Element with attribute	71

Chapter 1

Introduction

XQuery is a language developed by the World Wide Web Consortium(W3C) since 2001 to query Extensible Markup Language (XML), and since then many implementation achievements have been made. One of these achievements is the "Pathfinder" research project by the Database and Information Systems Department at the University of Constance¹, led by PD Dr. Torsten Grust and Jens Teubner.

The aim of Pathfinder is to implement XQuery as a query language and query XML data stored on back-end database systems. The architecture of Pathfinder is designed to adapt a number of different back-ends. One of these back-end adaptations is Monet, a main memory database system developed by the CWI² Institute for Mathematics and Computer Science and the University of Amsterdam³.

Another obvious back-end implementation are relational database management systems (RDBMS), due to fact that they have been populating the area of database systems for over 30 years and have been continuously improved. The power of RDBMS lies in their simple physical representation of tables of tuples of which sequential scans are supported perfectly from a low level view of computer hardware. On the other hand, if such sequential scans are not feasible, an efficient index is necessary.

This thesis presents a runnable implementation of the techniques to compile XQuery into Structured Query Language (SQL) which were developed by Grust, Sakr and Teubner [GST04]. The implementation uses a tree encoding which does not only support XQuery to SQL compilation needs but has been derived from former work on relational XPath implementations ([Gru02] and [GvKT03]) pro-

¹<http://www.inf.uni-konstanz.de/dbis>

²<http://monetdb.cwi.nl>

³<http://www.uva.nl>

viding full XPath functionality. This turns out to be essential seeing that beside the XQuery 1.0 specification of the W3C, XPath formerly was the starting point of XQuery development. Now XPath has been further developed towards XQuery needs and is known as XPath 2.0.

To gain a broader view of XQuery to SQL translation work, taking a look at other research papers is of course of great interest. With a classification of different approaches published in [KKN03], other publications [DTCO03] can be found which are very similar to the work of [GST04]. In fact, [DTCO03] was originally examined by the Pathfinder group due to its promising concept. All XML data of a document is stored in a single table instead of splitting up data into separate tables for each tuple as for example in [FK99], where the access to all tables in a database can be necessary. Also, a novel dynamic interval encoding which enables efficient computation on RDBMS was a further reason to take a closer view of [DTCO03]. The weakness of [DTCO03] however, is the lack of ability to compile arbitrary XQuery nesting and to actually produce results for a RDBMS.

In 1999, database research experienced a boost on publications dealing with XML storage and / or XML querying using RDBMS. A wide variety of solutions have been introduced making it rather intricate to distinguish between the different approaches and techniques.

Some work aims on scenarios using RDBMS to store and / or query XML, other take an opposite way of defining XML views from relational data and posing queries over these views. In [KKN03], the first scenario just mentioned is classified as XML Storage (XS) and the second as XML Publishing (XP). So the work described in [GST04] fits into the XS classification as does the Monet backend development.

Carrying on with the classification, focusing on XS now, finer categorisations were made towards schema-based (SB) or schema-oblivious (SO) approaches where again [GST04] can be counted as schema-oblivious. The strength of the schema-oblivious approach lies in the ability to store and query arbitrary XML documents conforming to the W3C specifications. For the last category, distinguishable differences concerning storage techniques were found and can be broadly classified as id-based, interval-based and path-based. The storage technique used by our XQuery to SQL implementation refers to the interval-based technique and will become more clear when we describe the XML document encoding in Chapter 3. Table 1.1 shows all the similarities between [GST04] and [DTCO03]. The work of [GST04], that appeared in 2004, has been additionally added. On the following page an overview of the classification is given together with Figure 1.1.

Technique	Scenario	Subproblems solved	Class of XML Schema considered	Class of XML queries handled
Monet	XS/SO	SS	all	-
Dynamic intervals [DTCO03]	XS/SO	QT	all	XQuery
<i>XQuery on SQL Hosts [GST04]</i>	<i>XS/SO</i>	<i>QT</i>	<i>all</i>	<i>XQuery</i>

XS/SO: XML storage, schema-oblivious QT: Query Translation , SS: Storage scheme

Table 1.1: An excerpt of relevant techniques by Krishnamurthy

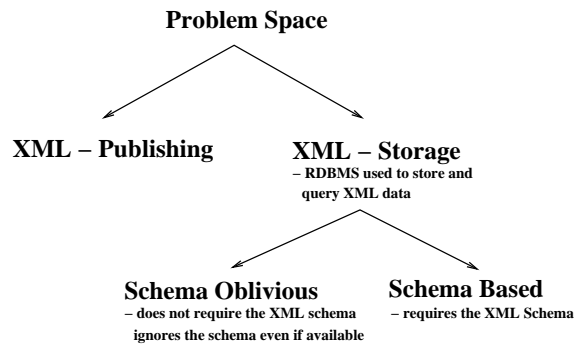


Figure 1.1: High-Level Taxonomy by Krishnamurthy

1.1 Aims

The aims of this thesis are to compile XQuery expressions into SQL and query XML documents on a RDBMS with a suitable tree encoding regarding performance times on different sizes of XML data. This work however, does not deal with updates of XML data.

The implementation works as a plugin for the Pathfinder project and generates the appropriate SQL statements. Since we plug into the Pathfinder XQuery compiler, issues like syntax checking or checks for semantical correctness are no concern for this work. The compiled SQL statements from the XQuery input can further be executed on a RDBMS to get a relational representation of the query result. With the result given back, a transformation back to XML text is feasible but is not subject of this work.

In other words, the aims are not to compile every kind of XQuery expression but to concentrate on the XQuery core subset containing only the most important expressions. The next step is to test if compilations are feasible, if the results are

correct and if the queries run within reasonable performance times. Certain problems occurring from the compiled SQL statements that produce duplicates due to multiple join operations will be addressed. Also, problems executing compiled SQL statements on a RDBMS will be dealt with in the performance measurements.

1.2 Structure of this Thesis

To successfully accomplish compilation of one language into another, it is first necessary to become accustomed to the source language structure and behaviour and to find ways of expressing its features in the target language - in this case to get familiar with the behaviour and language features of XQuery.

As the relational query language of SQL is well known, the differences of both query languages will be described and techniques and rules to translate these XQuery expressions to SQL as the target language will be presented.

The implementation is written in C, a programming language that is known to be highly efficient in low-level programming. For the central part of the compilation, an important technique called "pattern matching with *twig*" by [AGT89] is introduced. As Pathfinder compiles XQuery and provides the XQuery to SQL compiler with an abstract syntax tree representation of the input, a technique must be used to transfer the information from the XQuery abstract syntax tree to create an abstract syntax tree for SQL. Here, *twig* proves to be very useful. With its tree pattern matching technique, *twig* can find matches by dividing XQuery abstract syntax trees into single tree patterns according to its grammar and then generate SQL abstract syntax trees according to specified XQuery to SQL inference rules. The information can be passed from the XQuery side to the SQL code generation by *twig*-specific access pointers. Figure 1.2 shows the Pathfinder project together with the XQuery to SQL compiler.

As mentioned earlier, evaluating queries on a RDBMS for query performance is the last obstacle to overcome. The tests were made on an IBM DB2 database system with query time measurements on two different SQL variants, each on different sized XML documents.

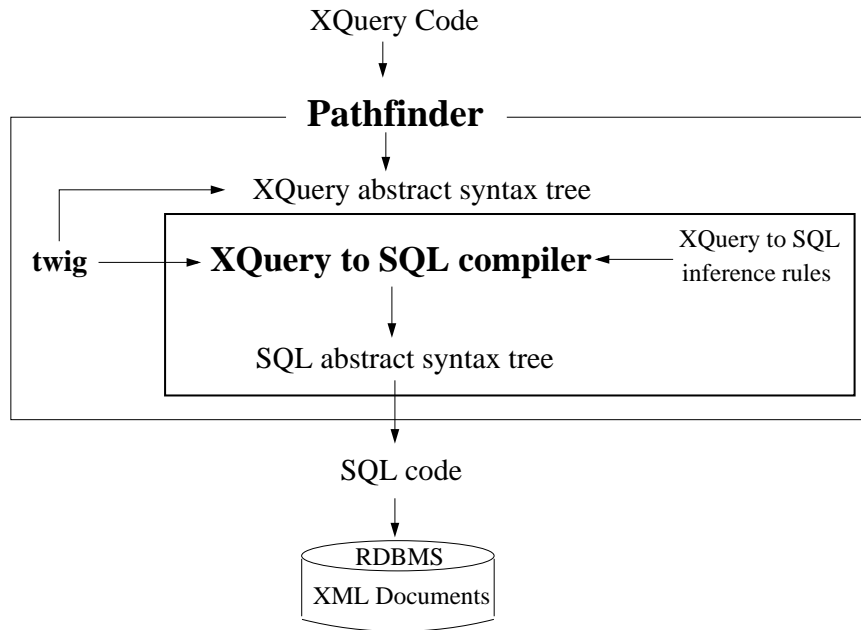


Figure 1.2: The Pathfinder Project with the XQuery to SQL Compiler

Chapter 2 is a refresher concerning XQuery and SQL and how to express XQuery behaviour in a relational way. Chapter 3 introduces the relational encoding of XML documents and results following the inference rules to compile single XQuery expressions into SQL. Chapter 4 handles the specific implementation using the *twig* compiler which accesses the XQuery abstract syntax tree to create an SQL abstract syntax tree. In Chapter 5 we execute the SQL code emitted by the compiler on an RDBMS to measure the performance of relational XQuery execution. Also, occurring problems are mentioned. In the last chapter a discussion on possible further developments is brought forth.

Chapter 2

XQuery and Relations

In this chapter an outline of the background of XQuery - its Data Model, related previous developments and the research work related to the Pathfinder development - will be given. Secondly, using tables towards XQuery expression nesting and ordering and the way to overcome these problems will be brought forth. Here, only the general idea of the techniques will be addressed and details will be explained in the next chapter.

2.1 An Introduction to XQuery

XQuery is derived from a functional language called Quilt¹. In such a functional language, every query is an expression which has to be evaluated. Expressions can be combined with other expressions to create new ones and every output of an expression can be an input for another expression. Expressions in functional languages can be evaluated in a parallel manner, not prescribing the order of evaluation, other than imperative languages that execute sequences of commands one by one.

Quilt provided the basis of XQuery as a functional language in which the for-let-where-orderby-return expression (FLWOR, pronounced: flower) is the most important one, although Quilt did not originally specify an "order-by" clause (FLWR). XQuery went further adding some new kinds of expressions such as "validate", "instance of" and "typeswitch". Quilt itself can be regarded as a more generalised language reflecting several other languages with the original idea of merging them together and forming a new one. Apart from the W3C specifications, the book "XQuery from the Experts" [CDFK03] gives an extensive view

¹http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html

of XQuery and related techniques. The information on basics and background is taken from this book.

A first and important part of the XQuery design is to specify a Data Model. This Data Model² was not defined according to XML text but in a more abstract form. Every document is regarded as a tree of nodes described as document-, element-, attribute-, text-, namespace-, processing instruction, and comment nodes. Moreover the data model also allows atomic values corresponding to the simple types of the W3C Recommendation "XML Schema, Part 2"³ in the form of integers, decimals, floats, doubles, dates, strings and booleans. Both, nodes and simple types, are referred to as items. Every input to a query is an instance of the Data Model as well as every output returned by a query. In XQuery, series of items are called sequences. Sequences can only hold nodes or simple types and cannot hold other sequences. A sequence $(a_0, (a_1, a_2))$ is equivalent to (a_0, a_1, a_2) .

Turning back to the FLWOR expression, which consists of clauses such as FOR, LET, WHERE, ORDER BY and RETURN, there is a close similarity to the SELECT FROM WHERE query block of SQL in a way that both expressions are used for selection and projection. A central feature of XQuery though, is the concept of variable bindings done in the FOR and LET clauses. The FOR clause iterates over a sequence of items and successively binds a variable $\$v_0$. LET binds a variable $\$v_0$ to an item or a whole list of items respectively. The WHERE clause acts in the same way as in SQL specifying a Boolean expression for every item. If the value is not true, the item is excluded from the variable bindings. The RETURN part gives back the rest of the items in form of a sequence. The Query example below depicts a typical FLWOR query incorporating XPath expressions evaluated on an XML document "books.xml".

```
FOR $b in doc("books.xml")//book
LET $c :=$b//author
WHERE count ($c) > 2
RETURN $b/title
```

A corresponding query in SQL would look like this:

```
SELECT books.title
FROM books
GROUP BY books.title
HAVING COUNT(books.author) > 2
```

The FOR clause binds every book node of book.xml and the LET clause binds a list of author nodes from each book. In the WHERE clause the number of authors is established by the built-in count() function, excluding the books having two

²<http://www.w3.org/TR/xpath-datamodel/>

³<http://www.w3.org/TR/xmlschema-2/>

or less than two authors. The RETURN clause gives back each title of the book nodes satisfying the WHERE clause.

The "books.xml" document looks like in Figure 2.1 (taken from [CDFK03]):

```
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="1992">
    <title>Advanced Programming in the UNIX Environment</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>65.95</price>
  </book>
</bib>
```

Figure 2.1: A fraction of the "books.xml" document

The result would look like this:

```
<title>Data on the Web</title>
```

As we can see, the query contains several XPath expressions using by // or / location steps. XQuery uses path expressions to locate nodes in XML data. This was mentioned here to show how tightly XPath and XQuery are connected. Path expressions will be picked up again later as we move towards the compilation part. To give an impression on the relationship between XQuery and SQL, the variable bindings can be seen as relations in which the relational table columns are the bound variables and the attribute values are the bound items. The LET clause becomes a further column extending a table, for example the previous query could have the bound variable \$c in the RETURN clause with:

```
RETURN $b/title,<authors>{count($c)}</authors>
```

giving back

```
<title>Data on the Web</title>
<authors>3</authors>
```

2.2 Iterative and Set Oriented Queries

The selection and projection of XQuery and SQL may seem very similar from the outside, but as described in [GST04], dealing with the structure of XML combined with flat relational tables is something that must be alluded. Another distinction between relational tables and XML, that must be pointed out, is the inherent relationship between nodes in a document and the necessity to express relationships between tables in form of SQL views. This also affects the relational XML encoding in a manner that not all relationships are directly present as attributes and have to be reconstructed via over relational joins.

Bringing up the Data Model features of XML again, relational database tables are considered to be like sets in a mathematical manner having no direct ordering other than those possibly derived from their values. XML data though has an intrinsic order and each node has a unique node identity. This order is not derived from its data values. Instead XQuery must retain the original ordering based on the source data.

This also accounts for sequences $(a_0, a_1 \dots a_n)$ which would be returned in a query

```
FOR $x in (1,2,3)
RETURN $x
```

in the exact order, namely 1,2,3. The concept is known as sequence order in XQuery

On the other hand, an SQL query like

```
SELECT rank, country, gold
FROM Athens2004
WHERE gold > 20
```

would not necessarily give back the three tuples ordered by its ranking, even if the tuples explain their ordering by themselves, but they must be explicitly defined by an ORDER BY rank clause.

rank	country	gold
1	USA	35
2	China	32
3	Russia	27

The problem also appears when nested FLWOR expressions occur and in this case the ordering becomes very important.

0	1		"	10	"	0	1		"	11	"
0	2		"	20	"	0	2		"	21	"
1	1		"	10	"	0	3		"	12	"
1	2		"	20	"	0	4		"	22	"

(a) lifted encoding (b) Flattened result
with the additions of
(1,2) and (10,20)

Figure 2.2: Lifted encoding of Query 2.1 and the result as a flattened table

Consider a query:

```
for $x in (1,2)
  return
    for $y in (10,20)
      return $x+$y
```

Query 2.1: A simple nested FLWOR

The result of this query is easy to compute by hand, taking the iteration of the first value bound to $\$x$ of the outer sequence, iterating through the inner sequence and adding each bound value of $\$y$ to the first bound value of $\$x$, receiving the values 11, 21, then returning to the outer iteration and performing the same procedure for the next value bound to $\$x$ and so on until we receive all 4 result values 11, 21, 12, 22.

The problem here is how to perform such a trivial query in a relational form. How can we "teach" a relational system to perform such iterations and which values are to be added to each other? An idea that comes to mind is to remember each iteration for every sequence of items and even every single item as a sequence of length one.

Looking again at the query, obviously two iterations must be stored for the outer and inner loops, in this case possibly numbered as 0,1 and 1,2. The idea here is related to the terms of loop lifting from [GST04]. If, for instance, a sequence like (10,20) in Query 2.1 is in an inner loop, then it must expose all iterations from the outer (0,1) loop together with its own iterations (1,2).

The nested loops are mapped into a small relation in Figure 2.2(a). The left column marks the outer iteration, the middle column the inner. At the end, a sequence of items should be present looking like in Figure 2.2(b), where the outer loop values, all set to 0, express a flattened interpretation of the nested query result. This

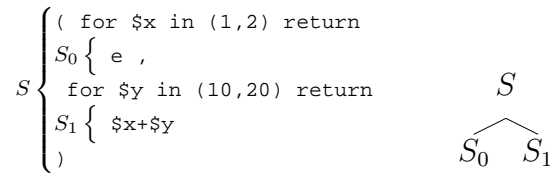


Figure 2.3: FLWOR scopes and resulting scope tree

example is very simple, but XQuery allows arbitrary nesting of expressions which makes it difficult to see how such nesting could be compiled without introducing scoping. In functional languages, variables with bound values can appear elsewhere in so called scopes. Such scopes are opened in FOR clauses setting a scope, but they also are opened by LET. We give a general notation for a FOR clause with *FOR* $\$v$ *IN* e_1 *RETURN* e_2 where e_1 and e_2 again can be FLWOR expressions.

Arbitrary nesting reveals tree shaped scope structures as depicted in Figure 2.3. Every bound variable is visible in a certain scope. The top scope S , however, holds no bound variables and just represents the root of the scope tree. Any occurrence of $\$x$ in expression e would mean that S_0 is its valid scope. The same applies to $\$y$ in the second FLWOR expression in the sequence, where S_1 is identified as its valid scope.

In S_1 though, only $\$y$ is in a valid scope and the occurrence of $\$x$ would yield back a scoping error. The scopes S_0 and S_1 are not nested, but separated by the top level sequence expression, so neither of them have any scoped variables in common. Again in S_0 , if a further FLWOR expression occurred, a child scope marked $S_{0,1}$ would be opened as a nested scope of S_0 . The notation $S_{x,y}$ with $x \in \{0, 1..\}^*$ and $y \in \{0, 1..\}$ identifies the child scope y of parent scope x .

In XQuery, expressions can contain variables bound in an enclosing scope. If an expression in a return clause is viewed on its own, some variables appear to be free. Take for instance the scopes of Query 2.1 in Figure 2.4 where the inner loop

$ S \left\{ \begin{array}{l} \text{for } \$x \text{ in (1,2) return} \\ S_0 \left\{ \begin{array}{l} \text{for } \$y \text{ in (10,20) return} \\ S_{0,1} \{ \$x+\$y \end{array} \right. \end{array} \right. $	<pre> ...return for \$y in (10,20) return \$x+\$y </pre>
--	--

Figure 2.4: Scopes on Query 2.1 the return expression, with $\$x+\y in scope $S_{0,1}$, both variables are considered to be free.

In other words, in scope $S_{0,1}$ we can say that a fixed value for each $\$x$ and $\$y$ can be established revealing the possible addition combinations to be returned in the correct sequence order, namely 1+10, 1+20, 2+10, 2+20.

To achieve this, a form of mapping must be provided. In the $S_{0.1}$ scope we have a mapping that is the same as the two left columns in Figure 2.2 (a) in the form of a relation called "map" with two columns "outer" and "inner" implementing the mapping of variables from scope S_0 to $S_{0.1}$. These mappings are created for every scope where a variable is recognised as free. Thus, to accomplish such nested expressions in a relational manner we need to generate relations with valid loop iterations for each expression resolving either in a sequence or single value as well as relations to facilitate the mapping from one scope to another.

To give an impression of how this works, we now run through Query 2.1 step by step and show the necessary mappings and scope-relations that must be performed.

Consider the sequence (1,2) already mapped into a relation, extended by its single positions in Table 2.2 (a). The compilation of such sequence expressions will be dealt with in the next chapter. This outer sequence (1,2) is transformed into a representation for scope S_0 exposing its iterations in Table 2.2 (b).

(a)	iter	pos	val	(b)	iter	pos	val
	0	1	"1"		1	1	"1"
	0	2	"2"		2	1	"2"

Table 2.2: The sequence of $\$x$ (a) and $\$x$ in scope S_0 (b)

With the opening of a new FLWOR and another scope, the map using $\$y$ in Table 2.3 (a) is made for S_0 to $S_{0.1}$ in Table 2.3 (b).

With this map, the relation of $\$y$ in $S_{0.1}$ is created by joining the map with $\$y$ over `outer=iter` and using the `inner` column of `map` as the new `iter`. The other columns of $\$y$ stay as they are - resulting in the relation in Table 2.3 (c).

(a)	iter	pos	val	(b)	outer	inner	(c)	iter	pos	val
	0	1	"10"		0	1		1	1	"10"
	0	2	"20"		0	2		1	2	"20"
								2	1	"10"
								2	2	"20"

Table 2.3: $\$y$ (a), the map (b) and $\$y$ in $S_{0.1}$ (c)

We can now prepare for $\$x$ in scope S_0 to be moved into scope $S_{0.1}$ by creating a map using the `iter` values of $\$y$ in $S_{0.1}$ from Table 2.3 (c) and numbering the `inner` column. Column numbering will be handled in the next chapter with concrete SQL statements.

The moving of relation $\$x$ in S_0 to $S_{0.1}$ is done by joining the `map` in Table 2.4 (b) with $\$x$ in S_0 (Table 2.4 (a)) over `outer=iter` to get the result in Table 2.4 (c). The `pos` values are all set to 0 describing the fully loop lifted relation.

(a) iter	pos	val	(b) outer	inner	(c) iter	pos	val
1	1	"1"	1	1	1	0	"1"
2	1	"2"	1	2	2	0	"1"
			2	3	3	0	"2"
			2	4	4	0	"2"

Table 2.4: $\$x$ in S_0 (a), the `map` for $\$x$ to $S_{0.1}$ (b) and the result for $\$x$ (c)

$\$y$ in $S_{0.1}$ can also be fully loop lifted in Table 2.5 (b), enabling the addition of the two sequences, Tables 2.5 (a) and (b) by means of a `join` over their `iter` values to Table 2.5(c).

(a) iter	pos	val	(b) iter	pos	val	(c) iter	pos	val
1	0	"1"	1	0	"10"	1	0	"11"
2	0	"1"	2	0	"20"	2	0	"21"
3	0	"2"	3	0	"10"	3	0	"12"
4	0	"2"	4	0	"20"	4	0	"22"

Table 2.5: Fully loop lifted $\$x$ (a) and $\$y$ (b) in $S_{0.1}$ and the addition result (c)

With the two sequences added up, a necessary back mapping is applied to flatten out the finished result. This is done by using the `map` in Table 2.4 (b) using the `outer` values as `iter` and numbering the `pos` column receiving the relation in Table 2.6 (b). As we have passed through two scopes, we have to apply a back mapping again to this result using the `map` of Table 2.3 (b) and end up with the flattened relation in Table 2.6 (c).

(a) iter	pos	val	(b) iter	pos	val	(c) iter	pos	val
1	0	"11"	1	1	"11"	0	1	"11"
2	0	"21"	1	2	"21"	0	2	"21"
3	0	"12"	2	3	"12"	0	3	"12"
4	0	"22"	2	4	"22"	0	4	"22"

Table 2.6: Back mapping of the result up to scope S

Every result returned from its scopes has to be mapped back into a flattened representation giving in this example the sequence (11,21,12,22).

Modifying the Query 2.1 the result could be again bound to a variable \$t as:

```
let $t :=
    for $x in (1,2)
        return
            for $y in (10,20)
                return $x+$y
return ...
```

Query 2.7: Query 2.1 with a further binding to \$t

With the result of the additions returned, \$t would be bound as

```
let $t:=(11,21,12,22) return ...
```

So the sequence would have to be flattened in a relational form in order to be further used in the return clause.

In the example of Query 2.1, not all map relations were shown and a "loop" relation was left out completely, as it is not necessary to understand the general idea. However, a loop relation plays a very important role and certainly becomes important if a FLWOR expression is given in the form of

```
for $x in (1,2) return "10"
```

where the variable \$x does not occur in the scope of the return clause but a loop relation is created from the sequence (1,2) and applied to the return expression resulting in a sequence ("10","10") which is mapped back the same way as in our example. All map and loop relations must be created for every FLWOR expression occurring in order to build a general rule for FLWOR translations.

Chapter 3

XQuery to SQL Translation Rules

3.1 The Result and Document Schemes

If XML data is to be stored in RDBMS, a form of ordering must be applied to represent the XML/XQuery concepts of document and sequence order. In the work of [Gru02] and [GvKT03] an indexing scheme was introduced by assigning pre and post values while parsing XML data and creating indexes of its document order. The XML fragment in Figure 3.1(a) and the indexed relational representation in Figure 3.1(b) make this clear. Here, only the `pre` index is of importance.

	<code>pre</code>	<code>node</code>	<code>iter</code>	<code>pos</code>	<code>pre</code>	<code>val</code>
<code><a></code>						
<code><c/></code>	1	<code><a></code>	0	1	NULL	"1.0"
<code><d/></code>	2	<code></code>	0	2	NULL	"x"
<code><e/></code>	3	<code><c></code>	0	3	0	NULL
<code></code>	4	<code><d></code>	0	4	3	NULL
	5	<code><e></code>				

(a) XML Fragment

(b) XML encoding

(c) Item encoding

Figure 3.1: Simple relational encodings of items

A simple encoding like this only incorporates nodes of tree fragments as relational tuples, but knowing about the Data Model, the question arises of how to encode simple types together with nodes. A sequence of arbitrary items $(1.0, "x", v, v')$, where v and v' are root nodes of two separate XML fragments, can be encoded in one single relation by introducing a further position

<pre> <a> <c/> </pre> <p>(a) Fragment 0</p>	<pre> <x> <y/>t </x> </pre> <p>(b) Fragment 1</p>
---	--

pre	size	level	kind	prop	frag
0	2	0	0	"a"	0
1	1	1	0	"b"	0
2	0	2	0	"c"	0
3	2	0	0	"x"	1
4	0	1	0	"y"	1
5	1	1	0	"b"	1
6	0	2	1	"t"	1

(c) Document encoding

Figure 3.2: The full relational encodings of XML documents / fragments

(*pos*) column. So the sequence looks like the table in Figure 3.1(c). Both nodes and simple types have positions in the *pos* column but differ in the storage in the *pre* and *val* columns - simple types, or better atomic values, have no nesting and therefore have a *NULL* as a *pre* value. Nodes, the other way round, have *NULL* as *val*. Here you would expect a node to have its node name in the *val* column. But that is not even necessary because *pre* values are unique according to an encoded XML document and thus may be referenced. In this sense the former description brings forth that there must be two different relations, one containing arbitrary items and the other containing detailed information on content and structure of a document. We refer to these encodings as relational "result" and "document" schemes.

Turning to the document schema more information is required than in 3.1(b). The recent work of [Gru02] has produced a variant of the encoding for documents moving the mapping scheme from a *pre/post* to a *pre/size* encoding but still use the previous techniques. The relational document encoding now holds the columns shown in Figure 3.2 (c).

With the *pre* value already mentioned, *size* gives us the size of a subtree below the node represented by this tuple. Size is regarded as an interval revealing the ability to scan for descendants of a context node. Furthermore, the *size* attribute is also insensitive to creation of subtree copies, being constant, whereas *post* would have to be re-calculated. This is where the term "interval" fits very well

into the interval-based classification of [KKN03].

The `level` attribute represents a node's depth in a tree starting from the root downwards and efficiently enables the location of child, parent and sibling nodes. The `kind` attribute corresponds to the Data Model concept of node kinds. The implementation though, only deals with text, node and document types. In `prop` the actual value, like tagname or text, is stored. The last attribute `frag` gives the information which document fragment a node belongs to. Nodes from different documents can appear in the database and can also be created on the fly by the element constructor causing possible subtree copies mentioned before.

What we hold now are two fundamental relational schemes used in the XQuery to SQL compilation, where the result schema is one of the most important ones, leading to final results or results for further computation, where the occurrence of necessary loop and mapping relations are applied.

3.2 Inference Rules

Having outlined relational representations to support XQuery expressions by introducing basic transformations in Chapter 2 and relational schemes in Section 3.1 we now move towards the more concrete SQL style. In [GST04] grammar rules were defined reflecting the core subset of the XQuery syntax:

(1)	<code>e ::= c</code>	atomic constants
(2)	<code> (e, e)</code>	sequences
(3)	<code> \$v</code>	variables
(4)	<code> let \$v := e return e</code>	let binding
(5)	<code> e/α::n</code>	path step (axis α, nodetest n)
(6)	<code> for \$v in e return e</code>	binding with iteration
(7)	<code> element t {e}</code>	element with tagname "t"

Figure 3.3: XQuery core expressions

Except for the variables, all the expressions on the right would be valid on their own according to the XQuery parser. The variable, obviously would be semantically incorrect lacking the presence of a valid scope prior to a binding by `for` or `let`.

All rules are given in a general form with:

$$\Gamma; loop; doc \vdash e \Rightarrow (q, doc')$$

- Γ specifies the current environment of XQuery variables mapped into a relational form. Environments can be seen as equivalent to scopes.
- $loop$ is the current representation of iterations.
- doc is the current encoded XML document.

This information is given in order to compile the expression e into an SQL statement q with a new created document doc' . We will see later on that only the element constructor can create new documents doc' . The compilation starts off with an empty environment $\Gamma = \emptyset$ and a singleton $loop$ table.

The expression e is run through an analysis in a top-down manner where every inference rule recognised has to be combined in a valid environment with a corresponding loop and doc relation. Each decomposed XQuery expression e_i corresponds to an inference rule and is translated into the respective SQL statement q_i and synthesised bottom-up to the single SQL statement q shown in Figure 3.4 below.

What must be commented here is that the top-down bottom-up traversal is in fact done in a more tree like manner, which is not shown in detail below.

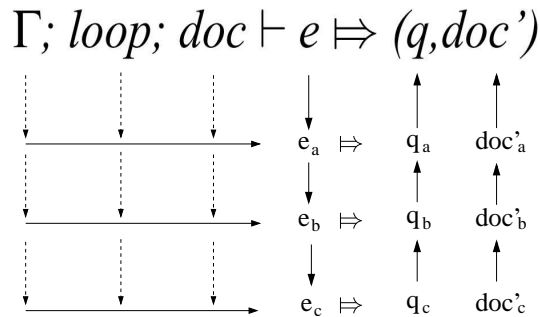


Figure 3.4: General top-down to bottom-up compilation

For simplicity, the seven inference rules have been divided into two categories: simple- and complex rules. The simple rules give an idea of how the loop relation is integrated, how orderings are guaranteed, and how an environment is applied. With this knowledge we can move on to the more complex rules in Section 3.2.2, where an extensive explanation is required.

The illustration of each inference rule is shown by its complete rule below a horizontal line with the resulting SQL transformation. Above, all the decomposable

expressions with their corresponding rules, SQL translations and intermediate relations are shown.

$$\frac{\Gamma; loop; doc \vdash e1 \Rightarrow (q1, doc') \quad \Gamma; loop; doc \vdash e2 \Rightarrow (q2, doc') \dots}{\Gamma; loop; doc \vdash E \Rightarrow (q, doc')}$$

3.2.1 Simple Rules

CONST

$$\Gamma; loop; doc \vdash c \Rightarrow \left(\begin{array}{l} \text{SELECT } 1.\text{iter}, 1 \text{ AS } \text{pos}, \\ \text{NULL AS } \text{pre}, c \text{ AS } \text{val} \quad , doc \\ \text{FROM } \text{loop AS } 1 \end{array} \right)$$

A constant is nothing but a value of a simple type. As the constant is standing on its own, only one iteration selected from the `loop` table stored on the database is assigned to the `iter` column. Its atomic value makes it a singleton item with the `pos` column given a constant value 1. We already know from Section 3.1 that atomic types hold `NULL` as `pre` and the `val` column holds its actual value. The document `doc` is not affected here.

SEQ

$$\frac{\Gamma; loop; doc \vdash e1 \Rightarrow (q1, doc') \quad \Gamma; loop; doc' \vdash e2 \Rightarrow (q2, doc'')}{\Gamma; loop; doc \vdash (e1, e2) \Rightarrow \left(\begin{array}{l} \text{SELECT } \text{iter}, \\ \text{e2.pos+m.pos AS } \text{pos}, \text{pre}, \text{val} \\ q1 \text{ UNION FROM } q2 \text{ AS } e2, \quad , doc'' \\ (\text{SELECT MAX(pos) AS } \text{pos} \\ \text{FROM } q1) \text{ AS } m \end{array} \right)}$$

A sequence is the next translation with two expressions to be translated. The inference rule is expressed as : $(e1, e2)$ below the horizontal line. The two rules above are the decomposed expressions with $e1$ as the head of the sequence and $e2$ as the tail.

A sequence $(10, 20, 30)$ is decomposed into $e1=10$ and $e2=(20, 30)$ to $(10, (20, 30))$.

- Expression $e1$ is translated by the *CONST* rule as "10" into $q1$
- Expression $e2$ is the tail sequence *SEQ* (20,30) nested as $q2$ in the SQL translation of the sequence rule.

Query $q1$ is merged with the UNION clause as well as $q2$.

An important part is the ordering of the given input sequence calculated by the pos of $q1$ and its $\text{MAX}(\text{pos})$ added to each of the pos values of $q2$.

- $\text{pos}(q1) = 1$,
- $\text{pos}(q2) = \text{pos}(q1') = 1$, $\text{pos}(q2') + \text{MAX}(\text{pos}(q1')) = 2$
with $q2 = (q1', q2')$
- The bottom-up synthesised pos values of the query give back
 $\text{pos}(q) = 1, 1 + \text{MAX}(\text{pos}(q1)), 2 + \text{MAX}(\text{pos}(q1)) = 1, 2, 3$

The SEQ inference rule gives back the relation of the input sequence as shown below.

iter	pos	pre	val
0	1	NULL	"10"
0	2	NULL	"20"
0	3	NULL	"30"

Table 3.1: The sequence (10,20,30) with calculated positions

VAR

$$\{\dots, \$v \mapsto qv, \dots\}; \text{loop}; \text{doc} \vdash \$v \Rightarrow (qv, \text{doc})$$

The VAR inference rule holds no explicit SQL query. Instead, the representation qv of variable $\$v$ is read from the environment Γ which was previously assigned to $\$v$ in a LET or FOR rule.

LET

$$\frac{\Gamma; \text{loop}; \text{doc} \vdash e1 \Rightarrow (q1, \text{doc}') \quad \Gamma + \{\dots, \$v \mapsto qv, \dots\}; \text{loop}; \text{doc}' \vdash e2 \Rightarrow (q2, \text{doc}'')}{\Gamma; \text{loop}; \text{doc} \vdash \text{let } \$v := e1 \text{ return } e2 \Rightarrow (q2, \text{doc}'')}$$

Now we can integrate the VAR rule into a LET rule. The $\$v$ is bound to an arbitrary expression $e1$ which can be compiled to any of the seven rules to be transformed to $q1$. The binding is shown below and the bound expression to an SQL query is presented in the left rule above the horizontal line. The variable binding of $\$v$ is now merged into the environment of the return expression $e2$. At this time of

top-down analysis we cannot say what kind of expressions $e1$ and $e2$ hold. The only process that is to be done here is to first enforce a compilation of $e1$ to $q1$ and then compile $e2$ to $q2$ where every occurrence of $\$v$ in $e2$ can be referenced to $q1$ and thus inserted into $q2$. A transformation of a *LET* inference rule is rather simple due to the fact that the whole expression $e1$ is bound to $\$v$ and can be referenced as a whole in $q2$. However, as we know from the XQuery basics, a `for` expression binds each tuple one by one which will turn out to be much more complex to translate, as we will now encounter when moving on to the complex inference rules.

3.2.2 Complex Rules

The last 3 rules give us the know-how to transform XPath *STEPS* into SQL with axis (α) and node tests (n), *FOR* rules with its iterations and according *loop* and *map* relations and finally create relational interpretations of `element` constructors (*ELEM*) where the document relation (*doc*) plays an important role.

STEP

$$\frac{\Gamma; \text{loop}; \text{doc} \vdash e \Rightarrow (qe, \text{doc}')}{\Gamma; \text{loop}; \text{doc} \vdash e / \alpha :: n \Rightarrow \left(\begin{array}{l} \text{SELECT DISTINCT } e.\text{iter}, d.\text{pre AS pos}, \\ \quad d.\text{pre}, \text{NULL AS val} \\ \text{FROM } qe \text{ AS } e, \text{doc}' \text{ AS } e', \text{doc}' \text{ AS } d \quad , \text{doc}' \\ \text{WHERE } e' . \text{pre} = e . \text{pre} \\ \quad \text{AND } e' . \text{frag} = e . \text{frag} \\ \quad \text{AND } \text{axis}(e', d, \alpha) \text{ AND } \text{test}(d, n) \end{array} \right)}$$

The *STEP* rule literally takes the last step in the path expression and translates the axis α with the help of the `pre`, `size` and `level` values and the node test n with `kind` and `prop`.

The next step is the expression e which can again be decomposed into a further step expression of the form $e / \alpha :: n$. We refer here to the term XPath step bundling: $(..((e / \alpha_1 :: n_1) / \alpha_2 :: n_2) / ..) / \alpha_k :: n_k$

The expression e can also be decomposed to an other type of expression other than a step for example `let $v := /alpha :: n`

All 13 axes from the XPath specification can be supported by `pre`, `size`, `level` calculations. Just consider a step `child::item` in an XML fragment:

```

<store>
  :
  <buyer></buyer>
  <buyer>
    <item></item>
  </buyer>
  <buyer>
    <item></item>
  </buyer>
</store>

```

Figure 3.5: An XML fragment

In the encoded Table 3.2 of Figure 3.5, the previous node in the former step expression e would give a location step from which point all `child` nodes named `item` are found by scanning all the nodes with `pre` values greater than the `pre` of the node in step e and smaller than the `pre` of e plus the `size` of e . Then again this scan is filtered for all nodes directly under the node in e with `level` of $e+1$ describing the child nodes. For the last condition the child nodes have to hold the `prop` value "item".

The concrete translation of $\text{axis}(e', d, \alpha)$ AND $\text{test}(d, n)$ is:

```

axis d.pre>e'.pre AND
    d.pre<=e'.pre+e'.size AND
    d.level=e'.level+1

```

```

test d.prop="item"

```

The first two conditions of the `WHERE` clause $e'.pre = e.pre$ and $e'.frag = e.frag$ make sure that the intermediate result of e with its `pre` values corresponds to the original document with alias e' . They also ensure that both queries are dealing with the same fragment.

The `STEP` rule does not really look that complex, due to its rather simple structural nesting of each step and easy to comprehend axis and node tests.

The `DISTINCT` keyword, necessary to remove duplicate nodes caused by the joins in the `FROM` clause, however, is a drawback of this implementation strategy. Operations such as `DISTINCT` and `ORDER BY` should not be used too generously because of their high performance costs on a RDBMS.

To point out this weakness, consider the document fragment in Table 3.2 with the `child::item` performed as the last step $\alpha_k::n_k$ and with the nodes `buyer` identified as a previous step giving the XPath expression `/buyer/child::item`.

pre	size	level	kind	prop	frag
.
10	1	2	0	"buyer"	0
11	0	3	0	"item"	0
12	1	2	0	"buyer"	0
13	0	3	0	"item"	0

Table 3.2: XML fragment of Figure 3.5 as a relation

The multiple joins of the document aliases `d`, `e'` and the result of the inner query `qe` as `e` would give back the result as in Table 3.2 without the `DISTINCT` keyword.

iter	pos	pre	val
0	11	11	NULL
0	13	13	NULL
0	11	11	NULL
0	13	13	NULL
0	11	11	NULL
0	13	13	NULL

Table 3.3: The duplicates created by multiple joins of Table 3.2

So `SELECT DISTINCT d.pre...` has to remove all duplicate tuples leaving only two with the `pre` values 11, 13. This of course is only a small example, but imagine finding several thousand `child::item` nodes. This problem gets worse, if the previous step `buyer` contained more than the two nodes with their `pre` values 10 and 12, but not having `child::item` nodes. They would also be joined as `e` with `e'` and `d` and create even more duplicates. These additional unnecessary results are referred to as redundant intermediate results displayed as a small XML tree in Figure 3.6. Only the `buyer` nodes with their `item` child nodes are of interest, but all `buyer` nodes are given back in the `buyer` step.

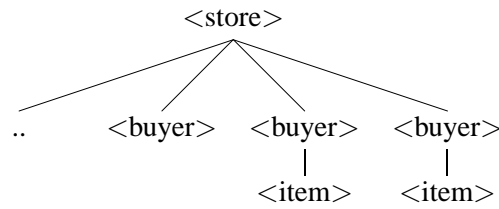


Figure 3.6: The tree representation of the XML fragment in Figure 3.5

A very effective development towards duplicate elimination is the Staircase Join by [GvKT03]. Using this technique a special join operator can be implemented by performing `pre` and `post` index calculations. Special algorithms of the Staircase Join can "prune" duplicate nodes in each path step before or during actual execution and skip nodes irrelevant to the path query. The Staircase Join can also be applied by performing calculations on the `pre / size` indexes of our implementation.

FOR

$$\begin{array}{c}
\{ \dots, \$v_i \mapsto qv_i, \dots \}; loop; doc \vdash e1 \Rightarrow (q1, doc') \quad \mathbf{loop}' \equiv (\text{SELECT iter FROM } \mathbf{qv}) \\
\mathbf{qv} \equiv \left(\begin{array}{l} \text{SELECT row() AS iter,} \\ \quad 1 \text{ AS pos, pre, val} \\ \text{FROM } q1 \\ \text{ORDER BY iter, pos} \end{array} \right) \quad \mathbf{map} \equiv \left(\begin{array}{l} \text{SELECT iter AS outer,} \\ \quad \text{row() AS inner} \\ \text{FROM } q1 \\ \text{ORDER BY iter, pos} \end{array} \right) \\
\left. \begin{array}{l} \text{SELECT inner AS iter,} \\ \quad \text{pos, pre, val} \\ \text{FROM } \mathbf{map}, \mathbf{qv}_i \\ \text{WHERE outer = iter} \end{array} \right\} + \{ \$v \mapsto \mathbf{qv} \}; \mathbf{loop}'; doc' \vdash e2 \Rightarrow (q2, doc'') \\
\hline
\{ \dots, \$v_i \mapsto qv_i, \dots \}; loop; doc \vdash \text{for } \$v \text{ in } e1 \text{ return } e2 \Rightarrow \\
\left(\begin{array}{l} \text{SELECT outer AS iter,} \\ \quad e2.iter * m.pos + e2.pos \text{ AS pos,} \\ \quad e2.pre, e2.val \\ \text{FROM } \mathbf{map}, q2 \text{ AS } e2, \\ \quad (\text{SELECT MAX(pos) AS pos} \\ \quad \text{FROM } q2) \text{ AS } m \\ \text{WHERE inner = e2.iter} \end{array} \right), doc''
\end{array}$$

The *FOR* rule above incorporates environment handling as well as the loop relation. In addition, the map relation introduced in Section 2.2 of Chapter 2 appears. This is not surprising, seeing that `map` is needed only in *FOR* rules.

Every time a *FOR* rule is encountered, a new environment must be created and the SQL expression created from e_1 with $e_1 \mapsto q1$ has to be converted into `qv` exposing each iteration. Then the `map` has to be created from `q1`. Further, a new `loop'` is created using `qv` for the new environment. The environment is a globally stored structure to be handled in more detail in the next chapter.

If an upper parent scope S_x exists, then there must be bound variables previously stored in the environment structure. These variables $\$v_i \mapsto qv_i$ are now free and must be re-mapped and stored back into the environment. Our new created `qv` is then also added to the environment. All mapped variables are now ready for access in e_2 . If a *FLWOR* expression is the top level expression, meaning that it is the first *FOR* rule reached in the top-down procedure with scope S , then qv_i is in fact `qv` itself. Hence, the environment is empty and there are no free variables to be mapped.

In the *FOR* rule a hypothetical function `row()` appears. This function is supposed to number columns such as `iter` or `pos`.

There are two ways of enabling such a numbering. Take for instance the `map` relation. By performing arithmetics on the `iter` and `pos` columns and creating a further small relation with `MAX(pos)` as shown in Figure 3.7, a numbering of the `inner` column

is easy to apply. The drawback of this solution is that the numbering is not dense in the `inner` column. With the query above and the relation `q1` in the left table of Figure 3.8, the mapping would hold the values shown in the `inner` column of Table 3.8 on the lower right.

<code>iter</code>	<code>pos</code>	.	.	<code>outer</code>	<code>inner</code>
1	1	.	.	1	5
1	2	.	.	1	6
2	1	.	.	2	9
2	2	.	.	2	10
2	3	.	.	2	11
3	1	.	.	3	15
3	2	.	.	3	16
3	3	.	.	3	17
3	4	.	.	3	18

Figure 3.8: Map relation without OLAP

The mapping in Figure 3.8 using `DENSE_RANK()` would now hold a dense ranking in the `inner` column numbered from 1 through to 9. The same function is applied to `iter` of the `qv` relation on the previous page.

It is not at all necessary to distinguish the iterations by `pos` but only by the `iter` columns. This dense ranking merely eliminates "holes" in `pos` or `iter` numbering and has no effect on the single it-

erations. The advantage of dense ranking its not so apparent here with such small numbers of iterations and `pos` values but it certainly turns out to have a greater effect the larger the results become creating even greater gaps without using this technique.

```
SELECT iter AS outer,
       iter*m.pos+e1.pos AS inner,
FROM q1 AS e1,
     (SELECT MAX(pos) AS pos
      FROM q1) AS m
ORDER BY iter,pos
```

Figure 3.7: Numbering without OLAP

The second way to overcome the problem of numbering columns is to make use of the Online Analytical Processing functionalities (OLAP) in Figure 3.9. Here the `DENSE_RANK()` function is of interest. In addition, `ORDER BY` is used, where the ordering is moved inside `DENSE_RANK()` to first order `iter` and `pos` of `q1` and then to perform a dense ranking on the `inner` col-

```
SELECT iter AS outer,
       DENSE_RANK() OVER( ORDER BY
                          q1.iter,q1.pos) AS inner,
FROM q1
```

Figure 3.9: `DENSE_RANK()` on the map relation

The consequence of absent dense ranking can lead to integer overflows on the SQL engine. High `iter` and `pos` values in an intermediate relation propagate even higher calculated inner values for the mapping which can again grow through further nesting of FLWOR Queries.

An important part, which has not been addressed so far, is the application of `DENSE_RANK()` to the resulting SQL query expression of the complete FOR rule. To densely rank the `pos` column instead of applying arithmetical expressions `e2.iter * m.pos + e2.pos AS pos`, an additional `PARTITION BY` is used to generate a dense `pos` column for each `iter` partition:

```
SELECT outer AS iter,
       DENSE_RANK()
         OVER(PARTITION BY map.outer
              ORDER BY map.inner, e2.pos) AS pos, e2.pre, e2.val
FROM map, q2 AS e2,
     (SELECT MAX(pos) AS pos
      FROM q2) AS m
WHERE inner = e2.iter
```

Moving back to the simple rules, for our *SEQ* rule, `PARTITION BY` can also be successfully implemented by introducing an extra `ord` attribute:

```
SELECT iter
       DENSE_RANK()
         OVER(PARTITION BY iter
              ORDER BY ord, pos) AS pos, pre, val
FROM SELECT *, 0 AS ord FROM q1
UNION
SELECT *, 1 AS ord FROM q2
```

The ordering by `ord, pos` makes sure that in a sequence with $(e1, e2) \Rightarrow (q1, q2)$ all items occurring in $q1$ ("a", "b") appear before items in $q2$ ("x", "y"):

The UNION creates

iter	pos	pre	val	ord
0	1	NULL	"a"	0
0	2	NULL	"b"	0
0	1	NULL	"x"	1
0	2	NULL	"y"	1

resolving to a sequence in correct order.

iter	pos	pre	val
0	1	NULL	"a"
0	2	NULL	"b"
0	3	NULL	"x"
0	4	NULL	"y"

ELEM

$$\begin{array}{c}
\Gamma; \text{loop}; \text{doc} \vdash e \Rightarrow (\mathbf{qe}, \text{doc}') \\
\mathbf{new\text{-}elems} \equiv \\
\left(\begin{array}{l}
\text{SELECT } e.\text{iter}, \text{dmax.pre} + \text{DENSE_RANK}() \text{ OVER} \\
\quad (\text{ORDER BY } \text{iter}, e.\text{pos}, e.\text{pre}) \text{ AS } \text{pre}, \\
\quad e.\text{size} \text{ AS } \text{size}, e.\text{level} \text{ AS } \text{level}, \\
\quad e.\text{kind} \text{ AS } \text{kind}, e.\text{prop} \text{ AS } \text{prop}, \\
\quad \text{dmax.frag} + \text{DENSE_RANK}() \text{ OVER} \\
\quad (\text{ORDER BY } \text{iter}) \text{ AS } \text{frag} \\
\text{FROM } (\mathbf{new\text{-}roots} \\
\quad \text{UNION} \\
\quad \text{SELECT } e1.\text{iter}, e1.\text{pos}, d2.\text{pre}, d2.\text{size}, \\
\quad d2.\text{level} - d1.\text{level} + 1 \text{ AS } \text{level}, d2.\text{kind}, d2.\text{prop} \\
\quad \text{FROM } \mathbf{qe} \text{ AS } e1, \text{document} \text{ AS } d1, \text{document} \text{ AS } d2 \\
\quad \text{WHERE } d1.\text{pre} = e1.\text{pre} \text{ AND } d2.\text{pre} \geq d1.\text{pre} \text{ AND} \\
\quad d2.\text{pre} \leq d1.\text{pre} + d1.\text{size} \\
\quad) \text{ AS } e, \\
\quad (\text{SELECT } \text{MAX}(\text{pre}) \text{ AS } \text{pre}, \text{MAX}(\text{frag}) \text{ AS } \text{frag}, \\
\quad \text{FROM } \text{document}) \text{ AS } \text{dmax}
\end{array} \right) \\
\mathbf{new\text{-}roots} \equiv \\
\left(\begin{array}{l}
\text{SELECT } \text{loop.iter}, 0 \text{ AS } \text{pos}, \\
\quad -2 \text{ AS } \text{pre}, \text{COALESCE}(\text{SUM}(\text{size}+1), 0) \text{ AS } \text{size}, \\
\quad 0 \text{ AS } \text{level}, 0 \text{ AS } \text{kind}, \mathbf{t} \text{ AS } \text{prop} \\
\text{FROM } \mathbf{qe} \text{ AS } e1 \text{ INNER JOIN } \text{document} \text{ AS } \text{doc} \\
\quad \text{ON } e1.\text{pre} = \text{doc.pre} \text{ RIGHT OUTER JOIN } \text{loop} \text{ AS } \text{loop} \text{ ON} \\
\quad \text{loop.iter} = e1.iter \\
\text{GROUP BY } \text{loop.iter}
\end{array} \right) \\
\hline
\Gamma; \text{loop}; \text{doc} \vdash \text{element } \mathbf{t} \{e\} \Rightarrow \left(\begin{array}{l}
\text{SELECT } \text{ne.iter}, 0 \text{ AS } \text{pos}, \\
\quad \text{ne.pre} \text{ AS } \text{pre}, \text{NULL} \text{ as } \text{val} \\
\text{FROM } \mathbf{new\text{-}elems} \text{ AS } \text{ne} \\
\text{WHERE } \text{ne.level} = 0
\end{array} \right)
\end{array}$$

The element constructor (*ELEM*) is the last inference rule rounding off the XQuery core syntax. Of course, further XQuery expressions like *if e1 then e2 else e3* could also be handled in form of an inference rule, but the aim is to achieve successful compilations of expressions to generally prove the feasibility of the XQuery to SQL compiler.

The *ELEM* rule has been modified in comparison to the rule in [GST04] using `DENSE_RANK()` as well as in the *SEQ* and *FOR* rules. Nevertheless, we will still be referring to the non-dense-ranked versions as well as to the dense-ranked when dealing with performance tests on a DB2 database system.

The verified *ELEM* rule has not been changed much towards the relational splitting between *new-roots* and *subtree-copies*. The former rule of [GST04] takes *new-roots* and *subtree-copies* and merges them together by the `UNION` operator in the outer query expression creating the new document fragment.

The new *ELEM* rule creates its *new-elements* by nesting the *new-roots* inside and performing a `UNION` with the copied nodes ($\Delta \subseteq \text{doc}$) out of the original persistent document: $\text{new-elements} = \text{new-roots} \cup \Delta$.

Taking a look at *new-roots*, every new root element has to be at the beginning of every new document fragment. That is why `pos` is set to 0 and `pre` to -2. The outer join ensures that for every `iter` value a tuple is generated, even if no child is present. If on the other hand a child is present, its `size` value is added up, otherwise the size value is assigned to 0.

```
(COALESCE(a,b) == IF a IS NULL then b else a).
```

In other words, `(COALESCE())` tests if a node is found from the path query `qe` by joining `qe` with `document` over their `pre` values. A document node is then found (not `NULL`) and 1 is added to the `size` of this node and assigned to the `size` of the new root node. If a node cannot be found from `qe` the join returns a `NULL` and the new root has 0 assigned to its `size` value.

In the second part of the `UNION` the *subtree-copies* (Δ) are created with their context nodes and their complete subtrees by the `pre` and `size` predicates.

Looking back at the document scheme all that is needed to be adjusted for the copied nodes are the `level` values. The `size` values do not even need to be touched at all proving the practical advantage of the `pre/size` over of the `pre/post` document encoding.

The outside query of *new-elements* performs a `DENSE_RANK()`, re-numbering new `pre` and adding new `frag` values by `dmax.frag + DENSE_RANK()...` to all nodes which can then be appended referring to the original document.

At the end, the whole result gives back all the new root nodes with `level=0`.

If it is necessary to access the document relation, all that must be done is to perform a `UNION` with the original document and the *new-elements* relation masking out the `iter` column.

```
SELECT* FROM document
UNION
SELECT pre,size,level,kind,prop,frag FROM new-elements
```

On the following page an example of a result and document relation with the el-

element construction $element\ y\ \{ / descendant::t / descendant::node() \}$ is shown. The path query qe results from the original document (frag=n) in Table 3.7 on the following page gives back:

iter	pos	pre	val
0	1	78	NULL
0	1	79	NULL

Table 3.4: The result from the path query

The new-roots relation would create Table 3.5 generating the size from the inner join with Table 3.7 (frag n) and qe .

iter	pos	pre	size	level	kind	prop
0	0	-2	2	0	0	"y"

Table 3.5: The new root \mathbf{y}

The final result gives back only the new root element with level=0 and frag n+1 taken from Table 3.7

iter	pos	pre	val
0	1	120	NULL

Table 3.6: The result schema of the new root \mathbf{y}

pre	size	level	kind	prop	frag
75	5	0	0	"x"	n
76	n
77	4	3	0	"t"	n
78	0	4	0	"u"	n
79	0	4	0	"v"	n
.	n
120	2	0	0	"y"	n+1
121	0	1	0	"u"	n+1
122	0	1	0	"v"	n+1

Table 3.7: The original document with frag n and the new document frag n+1

The documents relation is given back with all original elements in frag=n and all new elements in frag=n+1 together with its new root "y".

Chapter 4

XQuery to SQL Implementation

The following chapter explains the practical part with the knowledge gathered from the previous chapters - the schemes for expression nesting with maps and loops, the variable bindings and scopes. All is implemented according to the seven inference rules. For the implementation of the top-down decomposition and the resulting bottom-up synthesis of SQL the *twig* compiler is used and it is the main issue.

4.1 SQL Tree Expressions

At this point we could theoretically take an XQuery expression, break it down into appropriate inference rules, infer the SQL statements and nest all textual SQL fragments together to receive a ready to submit SQL translation. This would of course be very tedious and error-prone. Instead, we establish a common set of SQL expressions from all the seven inference rules. These expressions can then be divided into sub-expressions reflecting typical SQL statements and clauses in order to build up an internal tree representation.

More specifically, the first expression that would come to mind is the `SELECT FROM WHERE . . .` expression, which is regarded as the top-level of an SQL tree. Represented as a tree node, this top level SQL expression must hold at least a `SELECT-`, a `FROM-`, an optional `WHERE-`, and an optional `ORDER BY-` child. Hence, a node with a maximum of four child nodes can be necessary, but in fact, looking at the inference rules, only three child nodes are present, for example in the *FOR* inference rule `SELECT . . . FROM . . . ORDER BY` or `SELECT . . . FROM . . . WHERE`.

In the sequel, we describe the implementation and data structures that represent the SQL tree. We implemented each tree node by means of the `PFsnode_t` struct:

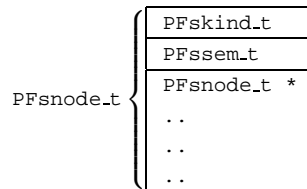


Figure 4.1: The structure type for SQL nodes

1. `PFskind_t` defines the kind of SQL node, for example a `SELECT FROM` node,
2. `PFssem_t` represents the nodes semantic value in form of a further structure holding strings, integers, floats, doubles, characters, booleans and an array of two strings for SQL relation-attribute references.
3. `PFsnode_t *` points to a child node, where only a maximum of 4 child nodes are possible as just mentioned.

Node kind `PFskind_t` is implemented as an enum type in C and holds every type of SQL node necessary. Corresponding to this enum, each node kind has a constructor function shown here in a more general form

*SQLnode * sql_node_constructor_name (SQLnode * child-node, ...).*

Each constructor returns a pointer to the new node, given the child nodes as arguments.

Once a node constructor is called, its node kind `PFskind_t` is passed down to a wiring function:

*SQLnode * wire[n](SQLnode-kind, SQLnode * child-node[1], ..., SQLnode * child-node[n])*

A wiring function connects ("wires") all the child SQL nodes created to its upper parent SQL node.

Each construction, decides which wiring function to call inside the constructor, depending on the number of child nodes `PFsnode_t *` an SQL node constructor holds. These wiring functions call lower wirings right down to leaf nodes in a left-first depth-first manner.

The implementation holds five wiring functions:

```
SQLnode * wire4 ( SQLnode-kind, SQLnode *n1, ...,SQLnode *n4 )
SQLnode * wire3 ( SQLnode-kind, SQLnode *n1, ... ,SQLnode *n3 )
SQLnode * wire2 ( SQLnode-kind, SQLnode *n1,SQLnode *n2 )
SQLnode * wire1 ( SQLnode-kind, SQLnode *n1 )
SQLnode * leaf( SQLnode-kind )
```

Take, for example, a simple addition of two attributes like in the SQL generation of the *SEQ* rule:

`e2.pos+m.pos,`

the construction would be :

```
plus ( attribute ("e2.pos"), attribute ("m.pos"))
```

First the attribute constructors would be called, both calling `leaf(attribute-kind)` passing their attribute node kinds in line 3.

```
1 SQLnode * attribute("e2.pos") / SQLnode * attribute("m.pos")
2 {
3   SQLnode *sql = leaf(attribute-kind);
4   sql->semantical-value = "e2.pos"; / sql->semantical-value = "m.pos";
5   return sql;
6 }

10 SQLnode * leaf(attribute-kind)
11 {
12   SQLnode *sql = allocate-memory(SQLnode);
13   sql->SQLnode-kind = attribute-kind;
14   sql->childnodes: set all 4 childnodes to 0
15   return sql;
16 }
```

Now, after returning back from the `leaf()` function, their nodes have been allocated with memory in line 12, the attribute kinds have been set in line 13 and all child nodes have been set to 0 in line 14. The semantical values `sql->semantical-value="e2.pos"` and `"m.pos"` can now each be assigned in the caller function `attribute()` in line 4.

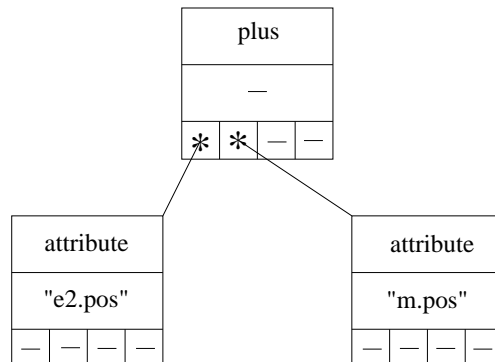


Figure 4.2: The storage structure of SQL nodes with wired child nodes

```

20 SQLnode * plus(n1,n2){
21   return wire2(plus,n1,n2)}

25 SQLnode * wire2(plus, n1,n2)
26 {
27   SQLnode *sql
28   sql = wire1 (plus, n1)
29   sql->child1 = n2
30   return sql
31 }

40 SQLnode * wire1(plus, n1)
41 {
42   SQLnode *sql
43   sql = leaf(plus)
44   sql->child0 = n1
45   return sql
46 }

```

The `plus()` constructor can be called (line 20), calling the wiring functions with `n1` and `n2` as the two memory allocated `attribute` nodes. The wirings are called right down to the `leaf()` function again where memory is allocated and its node kind `plus` is set. On return from each wiring function `wire1()` and `wire2()`, the `attribute` nodes `n1` and `n2` are wired to the `plus` node. Figure 4.2 shows the `plus` node with allocated memory and wirings. Mainly, only leaf nodes like attributes hold semantic values with string representations as their names. Certain special constructors such as the non-leaf `SELECT` node however, hold a boolean as a semantic value to set the `DISTINCT` keyword when creating the node for a *PATH* rule.

To make some constructions a bit more flexible in the number of nodes, a nesting with head and tail constructors were applied, especially for attributes in SELECT- and for relations in FROM clauses in a prefix parenthesised form:

attrib-list(attrib,attrib-list(...,nil))

constructs an attribute list for a SELECT clause and

relations(rel,relations(...,nil))

constructs a relations list for a FROM clause, both with *nil* at the end terminating the lists.

An example construction of a simplified SQL query

```
SELECT iter, pos, pre, val
FROM q1
```

would need the nesting of the following constructors :

```
sel-from-where (
  sel(false,
    attrib-list(attrib(str-literal("iter")),
      attrib-list(attrib(str-literal("pos")),
        attrib-list(attrib(str-literal("pre")),
          attrib-list(attrib(str-literal("val"))
            )),nil)
    )),nil)
  ),
  from(relations(rel(str-literal("q1")),nil)),
  nil,
  nil
)
```

These constructors would call for the *sel-from* constructor, a wiring function *wire4* (*sel-from-node, child1, child2, child3, child4*) with *child3* and *child4* set to *nil*.

The *sel-* and *from-* constructors would both call *wire1(sel-node,child1)* and *wire1(from-node,child1)* each with their node types. The child nodes *attrib-list* and *relations* would call *wire2(...)*

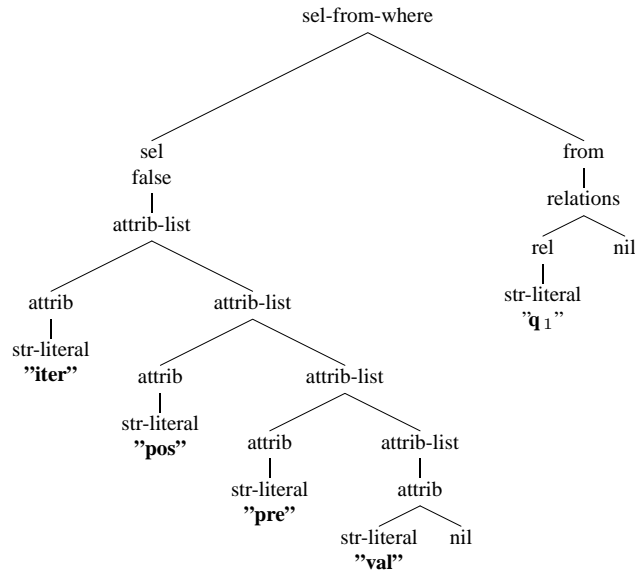


Figure 4.3: Resulting SQL tree

Figure 4.3 gives the abstract syntax tree for this nesting. For reasons of limited space, the two `nil` nodes for *child3* and *child4* have been left out in the tree. No information except for the node types and the semantic values are necessary. Any kind of syntactic "sugar" in form of SQL text is absent here and is generated right at the end of the compilation. Every inference rule can now be built up and we are ready for top-down handling on the XQuery side.

4.2 XQuery Pattern Matching

With a defined means to internally represent SQL, we can now face the actual compilation procedure. We first start off with a theoretical introduction to the *twig* compiler by [AGT89] and continue with the implementation using this technique in the second part.

4.2.1 An Introduction to the Twig Compiler

In the compilation procedure, *twig* runs through several phases to create an intermediate representation in form of a tree. Hence, compilation here is a procedure of a tree transformation. With both, the XQuery abstract syntax tree and the SQL sub-tree constructors we can use *twig* as a tool to create a compiler in order to transform an XQuery abstract syntax tree into an SQL abstract syntax tree. A basic and central part for this transformation are single rules of the *twig* compilation. They correspond to the tree grammar of the source tree and are given in the form of:

$$label_id : tree_pattern [\{ cost \}] [= \{ action \}]$$

Figure 4.4: The *twig* rule notation

For a rule to be valid it must hold at least a *label_id* and a *tree_pattern*. The $\{cost\}$ and $=\{action\}$ parts are optional, but without $=\{action\}$, no code generation with this specific rule would be performed. If the $\{cost\}$ part is left out, *twig* then returns a value `DEFAULT_COST`, a value that is mainly used in our implementation. The $\{cost\}$ and $=\{action\}$ parts contain the C code to perform the compilation.

Tree patterns, also called subject trees, are specified in a parenthesised prefix form, for example:

$$plus (e, plus (e, e))$$

The *twig* rules in Figure 4.5 with the cost and action parts left out, describe simple expressions with the *plus* operator.

```

expr: plus ( expr, expr )
expr: identifier
expr: constant

```

Figure 4.5: Twig rules for a plus expression

As already stated in Figure 4.4, the symbol left of ":" is referenced as a *label*, but it can also appear on the right side of a rule in the tree pattern. They are analogous to nonterminals in context free grammars. *Node_ids* are on the other hand only found on the right side. In the example, *identifier* and *constant* are regarded to be *Node_ids* being leaves in a tree sense, but *plus* is also a *Node_id* representing an internal tree node of the pattern.

Tree patterns can be described in a Backus-Naur-Form (BNF) as:

$$\begin{aligned} \textit{tree_pattern} & ::= \textit{node_id} \mid \textit{label_id} \mid \textit{node_id} (\textit{subtree_list}) \\ \textit{subtree_list} & ::= \textit{tree_pattern} \mid \textit{tree_pattern}, \textit{subtree_list} \end{aligned}$$

Twig requires both *node_id* and *label_id* identifiers to be declared before they are used. *Twig* assigns a unique integer value to every *node_id* and *label_id*. This id to integer mapping is given in a generated source file.

Moving over to the cost and action clauses, both are specified by enclosing C code in braces. The C code can be the form of any constructs. Together with the code, *twig* provides convenient access to the subject tree and user defined data structures.

1. $\$n\$$ denotes a pointer to a twig-internal data structure for the n th non-terminal leaf of a subject tree. It may be used for user-defined top-down processing for specific rules (e.g the *FOR* rule)
2. $\$\$$ denotes the root of a subject tree with a pointer to user defined structures
3. $\$n_1, n_2, \dots, n_{k-1}, n_k\$$ denotes a pointer to user defined structures of the n_k th child of the n_{k-1} th child of the n_{k-2} ... the n_1 th child of the root of a subject tree.

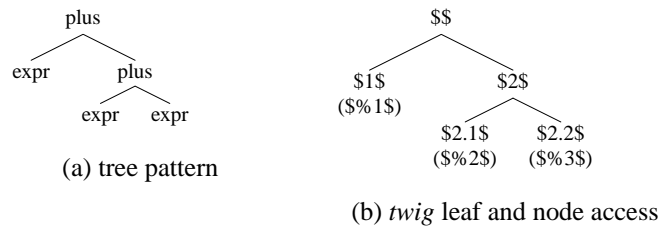


Figure 4.6: A tree pattern and its corresponding label access

Generally, *twig* handles trees in a bottom-up fashion (left to right), meaning that the tree in Figure 4.6 (a) would be handled by its node access pointers in Figure 4.6 (b), in the order \$1\$, \$2.1\$, \$2.2\$.

However, if a rule is specified as a so called top-down rule (keyword TOPDOWN in the cost part), then the order of tree handling is changed. The processing order has to be explicitly specified by the user. If, for instance, the leaf of \$2.1\$ has to be reduced before all other non-terminal leaves, the *twig* built in function `tDO($%2$)` has to be invoked inside the action code.

Reducing a non terminal means to execute the code in the action part of a corresponding twig rule. Note in Figure 4.6 (b) that the numbering of leaves for top-down are different than the general denotation. In top-down access, only the non-terminal leaves are of interest and are simply numbered from left to right.

If the pointer of `tDO($%2$)` of *expr* matches the rule: *expr: identifier* in Figure 4.5, then the *twig* access pointer points to the action code of this rule to be executed.

Recapitulating the idea of *twig*'s code generation, our implementation uses *twig* to express the XQuery to SQL translation rules. The tree pattern describes XQuery expressions in an abstract syntax form. *Twig* can access this information from the XQuery syntax tree to pass it to the SQL tree generation, or invoke further rule matching in a standard depth-first or top-down manner.

4.2.2 Twig Implementation

The *twig* input file is the core part of the XQuery to SQL compiler with both, source and target language, next to each other, broken up into rules in the *twig* syntax. Consequently, the number of rules to be implemented in the *twig* specification should correspond to the number of the seven inference rules presented in the previous chapter. This is only partially correct, because the *STEP* rule is additionally broken up into a step, root, an axis and a node-test rule.

The inference rules will not be dealt with in exactly the same order as in Chapter 3. We group some rules by their action code specifics. The rules *VAR*, *LET* and *FOR* have a special action code for environment handling. The *STEP* rule just mentioned has several rules to be examined and, together with *ELEM*, they access specific semantic information from the XQuery abstract syntax tree.

To give an introduction to the *twig* implementation, we start with the simple rules *CONST* and *SEQ*, followed by the rules *VAR*, *LET* and *FOR* and finally the *STEP* and *ELEM* with their additional rules and accesses to the abstract syntax tree.

To to give a better focus on the specific action code and structures, mostly small fractions of code will be displayed and explained step by step

At the starting point of compilation, we allocate an environment stack *env* to hold structure types *env_pair_t* as a pair of a variable (of type *PFvar_t*) and the corresponding tree node in form of an SQL node (type *PFsnode_t*).

$$\text{env_pair_t} \left\{ \begin{array}{l} \text{PFvar_t} \ * \\ \text{PFsnode_t} \ * \end{array} \right.$$

The environment stack is provided with the necessary functions to push, pop and look up pairs of the global *env* stack in the action code of a *twig* rule. The actual implementation of the stack allocation and access functions were taken and reused from an already implemented part of the Pathfinder framework.

In addition, the SQL nodes *loop*, *doc* and *sql* are initialised, and together with the environment stack, they are globally accessible.

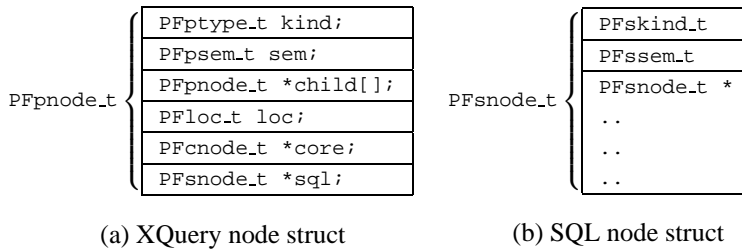


Figure 4.7: The C structure types for XQuery and SQL nodes

The abstract syntax tree is represented in Pathfinder as a tree of C structs (PFpnode_t), much like the internal SQL representation. Each node contains a field (sql) to hold the SQL equivalent for the XQuery expression rooted at this field. Figure 4.7 displays the structures of each side, with the newly introduced node structure of XQuery (Figure 4.7(a)) and the previously mentioned SQL node structure (Figure 4.7(b)).

The semantical value of any XQuery abstract syntax node can thus be accessed like:

```
($. . . $) ↦ sem.num
```

for the semantical integer value.

Twig pointers in a form of

```
($. . $) ↦ sql = sfw(sel(false..))
```

assign constructed SQL nodes of Figure 4.7 (b) to the sql field of the referenced subject tree node.

In fact, the implementation is of the form of

```
[[$. . $]] = sfw(sel(false..))
```

which is converted into

```
($. . $) ↦ sql = sfw(sel(false..))
```

by a macro expander,

but it is much more easy to understand the access to the XQuery tree information this way.

The PFpnode_t in Figure 4.7 (a) contains more information in its sem field, such as:

```
PFqname_t  qname;      /**< qualified name */
PFpaxis_t  axis;       /**< XPath axis */
PFpkind_t  kind;       /**< node kind */
```

This semantical information will be necessary to create SQL nodes for the *PATH* and *ELEM* inference rules. Only the sem field and the sql field will be needed from the XQuery side.

```

1 IntegerLiteral:    lit_int
2   {}
3   =
4   {
5       ($$)->sql =
6           sfw (
7               sel (false,
8                   attlist (attref ("1", "iter"),
9                           attlist (num (1),
10                                  attlist (null(),
11                                           attlist (str(int_to_str($$->sem.num)),
12                                                    ,nil ())
13                                           )
14                                   )
15                               tablerefs (tableref (loop, ident ("1"),nil()))),
16                               nil (),
17                               nil()
18                               );
19   };

```

Figure 4.8: The *CONST* inference rule in *twig*

Hence, the *twig* accesses of the XQuery abstract syntax tree can be used to pass the semantical values on to SQL constructors and then assign this constructed SQL tree node to the *twig* pointer access of the pointer (type `PFsnode_t`).

As an introduction to *twig* rules, we use the rather simple implementation of the *CONST* inference rule. Apart from its SQL node construction, no specific action- or cost code is applied, nor are any specially allocated structure variables other than `loop` (line 15 in Figure 4.8) and `sql` (line 5 in Figure 4.8) used or re-assigned.

The rule is given in line 1 as an `IntegerLiteral` and matches a `lit_int`. As `lit_int` is a terminal, it corresponds to an SQL node leaf having no more reductions.

We access the pattern root (`$$`) inside the SQL construction, and pass its semantical integer value $(\$\$) \mapsto \text{sem.num}$ in line 11 on to a string cast, and then is constructed as a string node to finally be placed in the relational `val` column of the SQL result schema as a `VARCHAR` type. The constructed SQL node `sql` is then assigned to the root $(\$\$) \mapsto \text{sql}$.

There is a *CONST* rule for each simple type:

```

IntegerLiteral:    lit_int
DecimalLiteral:   lit_dec
StringLiteral:    lit_str
:

```

Every constant has to be casted into a string to "fit" into the `val` column specified as `VARCHAR` of the SQL result relation.

is assigned to the `PFsnode_t` variable `seq` (lines 11 to 69).

Finally, the whole SQL node is assigned to the *twig* root SQL node in line 70.

At the end, these reductions inserted into the SQL node constructor would be ready to give back a complete SQL statement for a sequence.

To gradually build up the understanding of action code parts with bindings and occurrences of variables the *VAR* rule, to begin with, only consists of action code handling with the environment stack and an `sql` node.

```
1 Var_: var
2   = {
3     ( $$ )->sql= lookup_envpair($$->sem.var);
4   };
```

Figure 4.10: Action code for a variable occurrence.

Every occurrence of a variable in a valid scope implies that this variable is free and was bound to a distinct expression beforehand. Otherwise the XQuery compiler would return a non-bound variable- or scope-error and discontinue processing without the XQuery to SQL compiler even being invoked.

Variable usage is compiled as a simple environment lookup. Rules *LET* and *FOR* take care to maintain this environment with valid SQL representations for each bound variable.

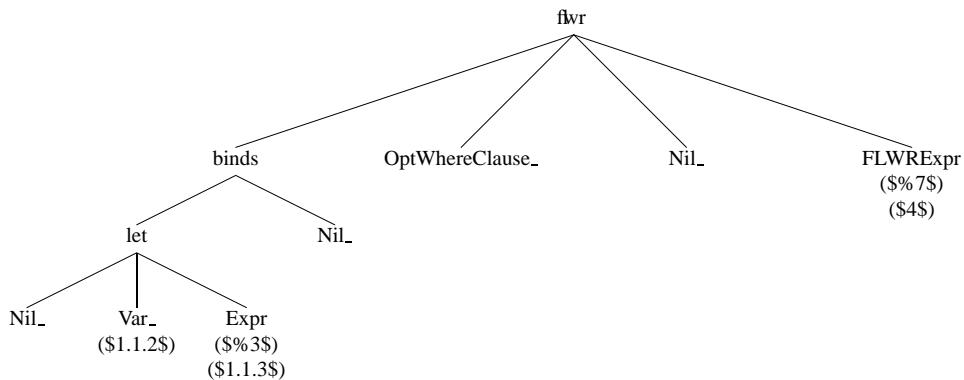


Figure 4.11: The LET tree pattern with *twig* access pointers

Let us first consider rule *LET* to exemplify this.

Figure 4.11 depicts the necessary access pointers. The *LET* rule is derived from the FLWOR expression, as well as the *FOR* rule. This is why the tree pattern of the *LET* rule looks more like a full FLWOR expression, but as we can see the *Nil_ node_ids* in the tree pattern are assigned non-*Nil_* leaves for another FLWOR variants.

So there are several *twig* rules with *FLWRExpr:..* to be matched, for example a tree pattern with an *order by* clause.

Now to take a look at the action code of the *LET* rule and the cost part, the first change towards the previously handled *twig* rules is that the cost part holds the keyword *TOPDOWN* in line 5 (see Figure 4.12). The cost part here tells *twig* to process this rule in a top-down fashion.

```

1 FLWRExpr: flwr (binds (let (Nil_, Var_, Expr), Nil_),
2                   OptWhereClause_,
3                   Nil_,
4                   FLWRExpr)
5   { TOPDOWN; }
6   =
7   {
8     int i;
9     env_pair_t pair;
10    PFarray_t *save_env = env;
11
12    tDO ($%3$)
13
14    env = new_environment();
15
16    for (i = 0; i < PFarray_last (save_env); i++)
17    {
18      pair =*((env_pair_t *) PFarray_at (save_env, i));
19      push_envpair(pair);
20    }
21    pair.var=($1.1.2$)->sem.var;
22    pair.node=($1.1.3$)->sql;
23    push_envpair(pair);
24
25    tDO ($%7$);
26
27    ( $$ )->sql = ($4$)->sql;
28    env = save_env;
29  };

```

Figure 4.12: LET binding of a variable together with environment handling.

We take a look at a let expression in XQuery syntax:

```
let $v:=e1 return e2
```

Before any occurrence of \$v appears in e₂, we must know what expression was bound to \$v as e₁. This is why a TOPDOWN is applied to the cost part. The non-terminal to be reduced first is e₁.

Looking at the tree pattern in Figure 4.11 the non-terminal Expr normally accessed as (\$1.1.3\$) must be passed to the twig built in function tDO() as %3\$ (line 12 of Figure 4.12). As this reduction means a variable assignment, an update to the global environment variable env is necessary. We first make a copy called save_env (line10).

```

11     PFarray_t *save_env = env;
12     :
14     env = new_environment();
15
16     for (i = 0; i < PFarray_last (save_env); i++)
17     {
18         pair =*((env_pair_t *) PFarray_at (save_env, i));
19         push_envpair(pair);
20     }

```

A new environment is constructed and assigned to `env` (line14). All the variables bindings are copied from `save_env` to `env` by pushing them one by one onto the stack again (lines 16 to 20). The `save_env` stack is assigned back to `env` at the end of the action code when moving back out of the environment scope of a `let` rule.

```

22     pair.var=($1.1.2$)->sem.var;
23     pair.node=($1.1.3$)->sql;
24     push_envpair(pair);

```

The new variable binding is finally added to the environment, and we are ready to compile the clause's return part.

```

26     tDO ($%7$);
27
28     ( $$ )->sql = ($4$)->sql;
29     env = save_env;
30 };

```

We invoke this compilation with `tDO($%7$)`, producing the overall expression result that we assign as `(4)→ sql` to the SQL node pointer of the *twig* root pointer `($$)→ sql` (line 28).

If e_1 was reduced top-down to a `lit_int: 10` and e_2 was then reduced top-down to an occurrence: `var` as for example: `let $v:=10 return $v`, then the result as `(4)→ sql` assigned to `($$)→ sql` in line 28 would be an SQL node of type constant matching the rule in Figure 4.8.

As mentioned earlier, the environment stack is restored in line 29 after leaving the scope of the `let` clause.

The *FOR* rule is the last rule that incorporates environment handling as well as the construction of the SQL nodes `loop'`, `map` and `qv`, and the access to the global variables `loop` and `sql`.

Also, a re-mapping of every previously stored SQL node-variable pair as: $\$v_i \mapsto q_{vi}$ from a parent environment scope to the new environment is necessary. The node constructions `loop'`, `map`, `qv` and `qvi` correspond to the inference rule *FOR* in Chapter 3.

The tree pattern of *FOR* is, as already mentioned, another variant of the FLWOR. The `bind` node in lines 1, 2 holds four child nodes, the previous `let` node holds only three child nodes. Like the rule *LET*, rule *FOR* is compiled in a top-down fashion.

```

1 FLWRExpr:    flwr (binds (bind (Nil_, OptPositionalVar_,
2                                Var_, Expr),
3                                Nil_),
4                                OptWhereClause_,
5                                Nil_,
6                                FLWRExpr)
:
:
12    PFarray_t *save_env = env;
15    PFsnode_t *save_loop = loop;
16    tDO($%4$);

```

Together with the global environment variable `env`, the global variable `loop` is copied to `save_loop` (line 15) before involving the reduction of the non-terminal leaf `Expr` in line 16.

The leaf `Expr` in the tree pattern corresponds to e_1 in the XQuery expression `for $v in e_1 return e_2` .

We can now construct the SQL representation `qv` of the newly bound variable.

Based on the result of e_1 's compilation, we create new iter values (using `DENSE_RANK()`), and set `pos` to 1 (See Rule *FOR* in Chapter 3)

```

20    qv =sfw(sel(false,
:
:
22    attlist(attref("q1","val"),nil()))),
23    tablerefs(tableref(($1.1.4$)->sql,ident("q1"),/* SQL:..FROM (SELECT..)AS q1*/
:
:

```

Then we create the scope mapping relation `map` with its outer and inner columns, using the result of e_1 .

```

28    map =sfw(sel(false,
:
:
31    tablerefs(tableref(($1.1.4$)->sql,ident("q1"),/* SQL:..FROM (SELECT..)AS q1*/
:
:

```



```

67             attlist(attref("vi","pos"),
68             attlist(attref("vi","pre"),
69             attlist(attref("vi","val"),nil())))),
70     tablerefs(tableref(map,ident("map"),
71             colnames(ident("outer"),
72             colnames(ident("inner"),nil()))),
73     tablerefs(tableref(
74             ((env_pair_t *) PFarray_at (save_env, i))->node,
75             ident("vi"),
76             colnames( ident("iter"),
77             colnames(ident("pos"),
78             colnames(ident("pre"),
79             colnames(ident("val"),
80             nil())))),
81     nil()),
82     eq(attref("map","outer"),attref("vi","iter")),
83     nil());
84     push_envpair(pair);
85 }

```

Each re-mapped `pair` is then pushed onto the new environment stack `env` in line 84.

We finally add the new variable binding to the environment.

```

87     pair.var=($1.1.3$)->sem.var;
88     pair.node=qv;
89     push_envpair(pair);

```

The environment is now set up to compile the pattern's return part:

```

100 tDO ($%8$);
101 env = save_env;
102 loop = save_loop;

```

The last two lines restore the environment variable `env` and relation `loop` after the reduction of `e2` in lines 101 and 102.

XPath steps are assembled from different tree patterns:

```
LocationPath_: root_
LocationPath_: locpath (LocationStep_, LocationPath_)
LocationStep_: step (NodeTest)
NodeTest: KindTest
NodeTest: NameTest
NameTest: namet
```

`LocationPath_: root_` matches the beginning of a rooted XPath expression and is implemented.

```
1 LocationPath_:    root_
2   = {
3       ($$)->sql = sfw(sel(true,
4           attlist(attref("l","iter"),
5           attlist(bigint(num(1)),
6           attlist(bigint(num(0)),
7           attlist(null()),
8           nil()))))
9       ),
10      tablerefs(tableref(ident("loop"),ident("l"),
11          colnames(
12              ident("iter"),
13              nil()
14          )),
15          nil()),
16          nil(),
17          nil()
18      );
19  };
```

The action code creates a result representation of the XML document root, taking its `iter` values from the `loop` relation (line 4), 1 for the `pos` column (line 5), 0 to the `pre` column (line 6) and finally `NULL` to its `val` column (line 7) resulting in the left hand table below.

iter	pos	pre	val	pre	size	level	kind	prop	frag
0	1	0	NULL	0	...	-1	4	"doc.xml"	1

The `pre` value of the result as 0 gives the reference to the document node of the XML document `doc.xml`. Its `size` value describes the number of sub-elements, and the original document fragment `frag = 1` in the above right table.

The rule `LocationPath_: locpath (LocationStep_, LocationPath_)` matches a location step that is processed by

```
1 LocationPath_:    locpath (LocationStep_, LocationPath_)
2   = {
3       ($$)->sql = sfw(
4           sel(true,
5           :
6           :
16          tablerefs(tableref(($2$)->sql,ident("e"),
17          :
18          :
25          and(eq(attref("e1","pre"),attref("e","pre"))),
```

```

26         and(eq(attref("e1","frag"),attref("d","frag")),
27             ($1$)->sql),
28         nil());
29     };

```

This fraction of the action code corresponds to the *STEP* inference rule. In line 4 the boolean value `true` sets the `DISTINCT` keyword in the `SELECT` clause. The `WHERE` clause holds the constraints to ensure that we are dealing with the same document fragment `frag` and `pre` value in lines 25, 26. The last parts of the `WHERE` clause in line 27 are axis and node test constraints retrieved from $(\$1\$) \mapsto \text{sql}$ (see below in lines 1 to 10).

Here, no top-down processing is necessary. The pattern matching of each location path `LocationPath_` and the `LocationStep_` cause a nesting of each single path step right down to the root. So the innermost expression is the root match: `LocationPath_: root_ .`

This recalls the path step bundling mentioned in the *STEP* inference rule in Chapter 3 with $(..((/\alpha_1::n_1) / \alpha_2::n_2) / ..) / \alpha_k::n_k$ where the `"/` on the innermost nesting is the match for the `root_.`

The non-terminal `LocationStep_` resolves into:

```

1 LocationStep_: step (NodeTest)
2 =
3 {
4   switch(($$)->sem.axis)
5   {
6     case p_child: ($$)->sql=and(
7       and(gt(attref("d","pre"),
8             attref("e1","pre")),
9           and(le(attref("d","pre"),
10              plus(attref("e1","pre"),
11                  attref("e1","size"))),
12              eq(attref("d","level"),
13                plus(attref("e1","level"),
14                    num(1))))))
15       ,($1$)->sql);
16     break;
17   }
18 }

```

$(\$\$) \mapsto \text{sem.axis}$ in line 4 contains the axis that this location step represents (α in $e/\alpha::n$ in the *STEP* inference rule). `axis` holds an enum type needed for the correct implementation of the step. If `p_child:` matches, then the `pre`, `size`, `level` calculations are specified in lines 7 to 14 to get the child node(s) for the context node(s) in `e`. The condition for the child axis of node `e1` would be:

`d.pre > e1.pre AND`

`d.pre <= e1.pre + e1.size AND`

`d.level = e1.level+1`

`d` representing the document `"doc.xml"` and `e1` the node(s) context set `e`. `d` has to be scanned in the interval from `e1.pre` to `e1.pre + e1.size` to get

all the descendant nodes of e_1 . Then those nodes from d are selected with $level+1=e.level$

pre	size	level	kind	prop	frag
20	3	2	0	"b"	0
pre	size	level	kind	prop	frag
18	5	0	0	"x"	0
19	2	2	0	"x"	0
20	3	2	0	"b"	0
21	1	3	0	"u"	0
22	0	4	0	"v"	0
23	0	3	0	"x"	0

The context set e is in the table above and its child nodes would be the nodes with the `pre` values 21 and 23 from the document in the lower table.

Analogous conditions are implemented for the other axes.

All 13 axes are specified as enum types. However, we implemented only the ancestor, parent, descendant and child axes, yet.

The last part to be matched is the n in $e/\alpha::n$, being the name test of a step. Three rules can match the name test with:

```
NodeTest: KindTest
```

```
NodeTest: NameTest
```

```
NameTest: namet
```

The `NodeTest: KindTest` rule also holds enum types to check for node types corresponding the XQuery specification:

```

1 NodeTest:    KindTest
2   =
3   {
4       switch($$->sem.kind)
5       {
6           case p_kind_node:    ($$)->sql=nil();
7                               break;
8           case p_kind_comment:break;
9           case p_kind_text:    ($$)->sql=eq(attref("d","kind"),num(1));
10                              break;
11          case p_kind_pi:      break;
12          case p_kind_doc:     break;
13          case p_kind_elem:    break;
14          case p_kind_attr:    break;
15      }
16 };

```

Only two of the seven node kinds have been implemented, being the most frequently used kind tests in XPath expressions. In the first case in line 6, `p_kind_node` corresponds to any step $e/\alpha::node()$. The `nil()` specifies that no SQL construct is necessary here. The second node kind $e/\alpha::text()$, where the SQL condition is that nodes in the document `d.kind` have the value 1. The value 1 is

coded as an arbitrary XML text node, and 0 an XML element node.

The last two rules are matched in case of a name test, for example $e/\alpha::item$.

NodeTest: NameTest

NameTest: namet

Here, the first rule NodeTest: NameTest would match. This rule performs no action code but passes on the SQL code upstream.

```

1 NameTest: namet
2   =
3   {
4     ($$)->sql=eq(attref("d","prop"),str($$->sem.qname.loc));
5   };

```

The second *twig* pattern actually implements the name test as the SQL condition $d.prop = n$ where n is the tag name to test for.

Let us review the XPath step bundling:

$$(..((/\alpha_1::n_1)/\alpha_2::n_2)/..)/\alpha_k::n_k$$

The *STEP* inference rule is implemented in this manner where the outermost SQL statement is the right step $\alpha_k::n_k$ and the innermost step is the root $"/$ left of $\alpha_1::n_1$.

So the root $"/$ gives all the nodes at the document root in the innermost SQL query below in lines 5, 6, 7 which are then filtered out step by step according to the "axis tests" and "node tests" of each step moving back to the outermost SQL query in lines 1 to 18. The DISTINCT keyword eliminates duplicate tuples caused by the joins of the nested query.

```

1 SELECT DISTINCT
2 FROM
3   (SELECT DISTINCT
4     FROM
5       (SELECT DISTINCT l.iter,0,0,"doc"
6         FROM           (loop) as l
7         ) AS e (iter,pos,pre,val),
8     doc AS e1,
9     doc AS d
10    WHERE e1.pre = e.pre AND
11          e1.frag = d.frag AND
12          "axis test" AND "node test"
13    ) AS e (iter,pos,pre,val),
14  doc AS e1,
15  doc AS d
16 WHERE e1.pre = e.pre AND
17       e1.frag = d.frag AND
18       "axis test" AND "node test"

```

The Element constructor denoted in the *ELEM* inference rule as

`element t {e}`

matches only the one rule below where `t` is the non-terminal `TagName` and `e` the non-terminal `ElementContent`.

In XQuery this would be like `element all-items{/descendant::items}`. The element name `all-items` refers to `TagName` and the XPath expression `/descendant::items` refers to `ElementContent`.

To begin with, the `new_elements` node must be constructed, where `new_roots` is nested directly inside. The `new_elements` construction implements the *ELEM* inference rule in Chapter 3. Here the *twig* action code has to pass the `TagName` to the `new_roots` construction from the XQuery semantical value in the abstract syntax tree (`((1)->sem.qname.loc` in line 38) giving the tag name of the element.

```

1 ElementConstructor:      elem (TagName, ElementContent)
2   new_elements=
3     sfw(sel(false,
4       :
5       :
6       :
7       :
8       :
9       :
10      :
11      :
12      :
13      :
14      :
15      :
16      :
17      :
18      :
19      :
20      :
21      :
22      :
23      :
24      :
25      :
26      :
27      :
28      :
29      :
30      :
31      :
32      :
33      :
34      :
35      :
36      :
37      :
38      :
39      :
40      :
41      :
42      :
43      :
44      :
45      :
46      :
47      :
48      :
49      :
50      :
51      :
52      :
53      :
54      :
55      :
56      :
57      :
58      :
59      :
60      :
61      :
62      :
63      :
64      :
65      :
66      :
67      :
68      :
69      :
70      :
71      :
72      :
73      :
74      :
75      :
76      :
77      :
78      :
79      :
80      :
81      :
82      :
83      :
84      :
85      :
86      :
87      :
88      :
89      :
90      :
91      :
92      :
93      :
94      :
95      :
96      :
97      :
98      :
99      :
100     :
101     :
102     :
103     :
104     :
105     :
106     :
107     :
108     :
109     :
110     :
111     :
112     :
113     :
114     :
115     :
116     :
117     :
118     :
119     :
120     :
121     :
122     :
123     :
124     :
125     :
126     :
127     :
128     :
129     :
130     :
131     :
132     :
133     :
134     :
135     :
136     :
137     :
138     :
139     :
140     :
141     :
142     :
143     :
144     :
145     :
146     :
147     :
148     :
149     :
150     :
151     :
152     :
153     :
154     :
155     :
156     :
157     :
158     :
159     :
160     :
161     :
162     :
163     :
164     :
165     :
166     :
167     :
168     :
169     :
170     :
171     :
172     :
173     :
174     :
175     :
176     :
177     :
178     :
179     :
180     :
181     :
182     :
183     :
184     :
185     :
186     :
187     :
188     :
189     :
190     :
191     :
192     :
193     :
194     :
195     :
196     :
197     :
198     :
199     :
200     :
201     :
202     :
203     :
204     :
205     :
206     :
207     :
208     :
209     :
210     :
211     :
212     :
213     :
214     :
215     :
216     :
217     :
218     :
219     :
220     :
221     :
222     :
223     :
224     :
225     :
226     :
227     :
228     :
229     :
230     :
231     :
232     :
233     :
234     :
235     :
236     :
237     :
238     :
239     :
240     :
241     :
242     :
243     :
244     :
245     :
246     :
247     :
248     :
249     :
250     :
251     :
252     :
253     :
254     :
255     :
256     :
257     :
258     :
259     :
260     :
261     :
262     :
263     :
264     :
265     :
266     :
267     :
268     :
269     :
270     :
271     :
272     :
273     :
274     :
275     :
276     :
277     :
278     :
279     :
280     :
281     :
282     :
283     :
284     :
285     :
286     :
287     :
288     :
289     :
290     :
291     :
292     :
293     :
294     :
295     :
296     :
297     :
298     :
299     :
300     :
301     :
302     :
303     :
304     :
305     :
306     :
307     :
308     :
309     :
310     :
311     :
312     :
313     :
314     :
315     :
316     :
317     :
318     :
319     :
320     :
321     :
322     :
323     :
324     :
325     :
326     :
327     :
328     :
329     :
330     :
331     :
332     :
333     :
334     :
335     :
336     :
337     :
338     :
339     :
340     :
341     :
342     :
343     :
344     :
345     :
346     :
347     :
348     :
349     :
350     :
351     :
352     :
353     :
354     :
355     :
356     :
357     :
358     :
359     :
360     :
361     :
362     :
363     :
364     :
365     :
366     :
367     :
368     :
369     :
370     :
371     :
372     :
373     :
374     :
375     :
376     :
377     :
378     :
379     :
380     :
381     :
382     :
383     :
384     :
385     :
386     :
387     :
388     :
389     :
390     :
391     :
392     :
393     :
394     :
395     :
396     :
397     :
398     :
399     :
400     :
401     :
402     :
403     :
404     :
405     :
406     :
407     :
408     :
409     :
410     :
411     :
412     :
413     :
414     :
415     :
416     :
417     :
418     :
419     :
420     :
421     :
422     :
423     :
424     :
425     :
426     :
427     :
428     :
429     :
430     :
431     :
432     :
433     :
434     :
435     :
436     :
437     :
438     :
439     :
440     :
441     :
442     :
443     :
444     :
445     :
446     :
447     :
448     :
449     :
450     :
451     :
452     :
453     :
454     :
455     :
456     :
457     :
458     :
459     :
460     :
461     :
462     :
463     :
464     :
465     :
466     :
467     :
468     :
469     :
470     :
471     :
472     :
473     :
474     :
475     :
476     :
477     :
478     :
479     :
480     :
481     :
482     :
483     :
484     :
485     :
486     :
487     :
488     :
489     :
490     :
491     :
492     :
493     :
494     :
495     :
496     :
497     :
498     :
499     :
500     :
501     :
502     :
503     :
504     :
505     :
506     :
507     :
508     :
509     :
510     :
511     :
512     :
513     :
514     :
515     :
516     :
517     :
518     :
519     :
520     :
521     :
522     :
523     :
524     :
525     :
526     :
527     :
528     :
529     :
530     :
531     :
532     :
533     :
534     :
535     :
536     :
537     :
538     :
539     :
540     :
541     :
542     :
543     :
544     :
545     :
546     :
547     :
548     :
549     :
550     :
551     :
552     :
553     :
554     :
555     :
556     :
557     :
558     :
559     :
560     :
561     :
562     :
563     :
564     :
565     :
566     :
567     :
568     :
569     :
570     :
571     :
572     :
573     :
574     :
575     :
576     :
577     :
578     :
579     :
580     :
581     :
582     :
583     :
584     :
585     :
586     :
587     :
588     :
589     :
590     :
591     :
592     :
593     :
594     :
595     :
596     :
597     :
598     :
599     :
600     :
601     :
602     :
603     :
604     :
605     :
606     :
607     :
608     :
609     :
610     :
611     :
612     :
613     :
614     :
615     :
616     :
617     :
618     :
619     :
620     :
621     :
622     :
623     :
624     :
625     :
626     :
627     :
628     :
629     :
630     :
631     :
632     :
633     :
634     :
635     :
636     :
637     :
638     :
639     :
640     :
641     :
642     :
643     :
644     :
645     :
646     :
647     :
648     :
649     :
650     :
651     :
652     :
653     :
654     :
655     :
656     :
657     :
658     :
659     :
660     :
661     :
662     :
663     :
664     :
665     :
666     :
667     :
668     :
669     :
670     :
671     :
672     :
673     :
674     :
675     :
676     :
677     :
678     :
679     :
680     :
681     :
682     :
683     :
684     :
685     :
686     :
687     :
688     :
689     :
690     :
691     :
692     :
693     :
694     :
695     :
696     :
697     :
698     :
699     :
700     :
701     :
702     :
703     :
704     :
705     :
706     :
707     :
708     :
709     :
710     :
711     :
712     :
713     :
714     :
715     :
716     :
717     :
718     :
719     :
720     :
721     :
722     :
723     :
724     :
725     :
726     :
727     :
728     :
729     :
730     :
731     :
732     :
733     :
734     :
735     :
736     :
737     :
738     :
739     :
740     :
741     :
742     :
743     :
744     :
745     :
746     :
747     :
748     :
749     :
750     :
751     :
752     :
753     :
754     :
755     :
756     :
757     :
758     :
759     :
760     :
761     :
762     :
763     :
764     :
765     :
766     :
767     :
768     :
769     :
770     :
771     :
772     :
773     :
774     :
775     :
776     :
777     :
778     :
779     :
780     :
781     :
782     :
783     :
784     :
785     :
786     :
787     :
788     :
789     :
790     :
791     :
792     :
793     :
794     :
795     :
796     :
797     :
798     :
799     :
800     :
801     :
802     :
803     :
804     :
805     :
806     :
807     :
808     :
809     :
810     :
811     :
812     :
813     :
814     :
815     :
816     :
817     :
818     :
819     :
820     :
821     :
822     :
823     :
824     :
825     :
826     :
827     :
828     :
829     :
830     :
831     :
832     :
833     :
834     :
835     :
836     :
837     :
838     :
839     :
840     :
841     :
842     :
843     :
844     :
845     :
846     :
847     :
848     :
849     :
850     :
851     :
852     :
853     :
854     :
855     :
856     :
857     :
858     :
859     :
860     :
861     :
862     :
863     :
864     :
865     :
866     :
867     :
868     :
869     :
870     :
871     :
872     :
873     :
874     :
875     :
876     :
877     :
878     :
879     :
880     :
881     :
882     :
883     :
884     :
885     :
886     :
887     :
888     :
889     :
890     :
891     :
892     :
893     :
894     :
895     :
896     :
897     :
898     :
899     :
900     :
901     :
902     :
903     :
904     :
905     :
906     :
907     :
908     :
909     :
910     :
911     :
912     :
913     :
914     :
915     :
916     :
917     :
918     :
919     :
920     :
921     :
922     :
923     :
924     :
925     :
926     :
927     :
928     :
929     :
930     :
931     :
932     :
933     :
934     :
935     :
936     :
937     :
938     :
939     :
940     :
941     :
942     :
943     :
944     :
945     :
946     :
947     :
948     :
949     :
950     :
951     :
952     :
953     :
954     :
955     :
956     :
957     :
958     :
959     :
960     :
961     :
962     :
963     :
964     :
965     :
966     :
967     :
968     :
969     :
970     :
971     :
972     :
973     :
974     :
975     :
976     :
977     :
978     :
979     :
980     :
981     :
982     :
983     :
984     :
985     :
986     :
987     :
988     :
989     :
990     :
991     :
992     :
993     :
994     :
995     :
996     :
997     :
998     :
999     :
1000    :

```

In the nested `new_roots` construction, lines 25 to 40 . . . , the `ElementContent` has to be reduced similar to the XPath step `descendant-or-self::node()`. (`2`)`->sql` in line 40.

```
41         tablerefs(tableref(($2$)->sql,ident("e1"),
42
43 ($$)->sql=
44     sfw(
45         sel(false,
46             attlist(attref("ne","iter"),
47                 attlist(num(0),
48                     attlist(attref("ne","pre"),
49                         attlist(cast(t_subst(null(),varchar(num(SQL_TYPE_SIZE))))),
50                     nil())))),
51         tablerefs(tableref(new_elements,ident("ne"),
52             colnames(ident("iter"),
53                 colnames(ident("pre"),
54                     colnames(ident("size"),
55                         colnames(ident("level"),
56                             colnames(ident("kind"),
57                                 colnames(ident("prop"),
58                                     colnames(ident("frag"),
59                                         nil())))))))),
60         nil()),
61         eq(attref("ne","level"),num(0)),
62         nil()
63     );
```

Finally, in lines 44 to 62 the `new_elements` construction is nested inside the whole SQL expression in line 51 and returned as the pattern's SQL equivalent (see Rule *ELEM* in Chapter 3).

Printing SQL

The implemented *twig* code handles the compilation to SQL for a large subset of XQuery.

The XQuery-to-SQL compilation is invoked by Pathfinder with the command

```
pf -Sc
```

instructing Pathfinder (*pf*) to compile XQuery into SQL (parameter *-S*) and to stop after compilation (parameter *c*). The XQuery expression is read from *stdin*.

The SQL compilation is started by the function:

```
PFsql_gen (PFpnode_t *n)
```

The compilation starts with an empty environment `env = new_environment();`

The `loop` and `doc` pointers are initialised to the two tables "loop" and "doc" (`loop = table ("loop");` and `doc = table ("document");`). Both are available as persistent tables in the back-end RDBMS.

The *twig* compiler is then invoked with

```
sql = (rewrite (n, 0) )->sql;
```

passing the `PFpnode_t n` as the root of the XQuery abstract syntax tree to the `rewrite` function of *twig*. The result of the *twig* compilation is a pointer to the fully synthesised SQL abstract syntax tree which is assigned to the `sql` pointer.

Function `sql_print (PFsnode_t sqlroot);` finally serializes the SQL abstract syntax tree into a textual form.

```

1  sql_print (PFsnode_t *n) {
2
3  switch (n->kind)
4  {
5  case s_sfw_expr:    PFprettyprintf ("%c(SELECT ", START_BLOCK);
6                    sql_print (n->child[0]);
7                    PFprettyprintf (" FROM ");
8                    sql_print (n->child[1]);
9                    if(n->child2->kind != s_nil){
10                   PFprettyprintf (" WHERE ");
11                   sql_print (n->child[2]);
12                   }
13                   if(n->child3->kind != s_nil){
14                   sql_print (n->child[3]);
15                   }
16                   PFprettyprintf ("%c", END_BLOCK);
17                   break;
18                   :
19                   :
50  case s_lit_str:    PFprettyprintf ("'%s'", n->sem.str);
51                   break;

```

This function walks through the the SQL abstract syntax tree in a recursive manner and produces SQL code by printing the SQL syntax for any of the node kinds in the abstract syntax tree. The first node encountered is the "SELECT FROM WHERE" node with its node kind (`s_sfw_expr`) in line 5. The `PFprettyprintf` function prints SQL code corresponding to this node kind. The `sql_print` clause

is then called recursively for `n→child[0]` attributes of `SELECT`. Before optional nodes like the `WHERE` clause in line 9 are printed and further processed they have to be checked if they are not of kind `s_nil`. Semantical values such as nodes of kind `s_lit_str`: are printed by `PFprettyprintf` as shown in line 50.

Chapter 5

Performance Tests

5.1 Testing Procedures

To test if compiled XQuery expressions can be successfully executed on a RDBMS, seven XML documents were generated using the XML generator tool XMLgen.:

```
<site>
  <regions>
    <africa>
      <item id="item0">..</item>
    </africa>
    <asia>
      <item id="item1">..</item>
      <item id="item2">..</item>
    </asia>
    <europe>
      :
      :
    </europe>
      :
      :
  </regions>
  <categories>
    :
  </categories>
</site>
```

Figure 5.1: A fraction of a document from XMLgen

The documents that were generated varied in sizes between 100 KB and 1.1 GB and were encoded and stored on the server `phobos29` hosting an IBM DB2 UDB V8.1 database system. The hardware used to support the RDBMS consisted of a dual 2.2 GHz Pentium 4 Xeon system with 2 GB of RAM and Linux as the operating system with kernel version 2.4.

Pathfinder was not directly connected to the database system, so the XQuery input had to be compiled, and instead of sending the output directly to stdout they were each redirected into single .sql files:

```
Pf -Sc > sql-query.sql
```

```
for $x in (1,2,3) ....
```

A client connection to the database system was established and the .sql files were passed to the database input by:

```
db2batch -d xml -f sql-input-file.sql -i complete
```

The db2batch command was executed using -d xml, with xml being the database, holding the encoded documents: A.document, B.document up to G.document. The table A.document represents the smallest XML document A.xml (113 KB) and G.document the largest document (1 GB).

As described in the previous chapter invoking SQL printing, doc was constructed with the name "document" and had to be replaced in each SQL file with "A.document", "B.document" etc.

Due to the fact that a query had to be performed on all seven document sizes, with two representations, namely the document and the result schemes and DENSE_RANK() and non-DENSE_RANK() SQL variants, all in all 28 different SQL files were run on the database for one single compiled XQuery input.

A macro #define DENSE_RANK 1 was set in the twig implementation and in the action code of the SEQ, FOR and ELEM rules, DENSE_RANK() and non-DENSE_RANK() SQL node constructions were implemented. On setting the macro to 1, the DENSE_RANK() variants are constructed, if the macro is not set to 1 the non-DENSE_RANK() version is set when recompiling Pathfinder.

With the parameter -f of db2batch the input file is given and -i complete instructs the database system to give back all time measurements for the query input and the number of rows retrieved.

```
Number of rows retrieved is:  
Number of rows sent to output is:  
Prepare Time is:  
Execute Time is:  
Fetch Time is:  
Elapsed Time is:
```

The output time for the retrieved rows, however, is not recorded in the time measurements and is irrelevant for the testing. Each query was executed four times and only the last three time measurements were recorded to ensure a cached database buffer access. The three Execute Time results for each query were then averaged.

5.2 Occurring Problems

Two major boundaries caused two problems trying to set off an arbitrary nesting of FLWOR expressions together with element constructors. Originally, the idea was to form a test query with all the inference rules also incorporating nesting, but the compiled SQL code reached sizes up to 100 K which the RDBMS could not parse due to its restricted input of 65 K. The code tended to "explode" in size whenever nesting was applied. Even removing unnecessary whitespaces from the SQL files did not help sufficiently. The second boundary was caused by integer overflow due to the non-DENSE_RANK () queries mentioned in the *FOR* inference rule in Chapter 3.

Nevertheless, in order to prove the correctness of the XQuery to SQL compiler, single XQuery expressions corresponding to SEQ, FOR and ELEM inference rules were tested separately only using the DENSE_RANK () version. Finally, one XQuery Expression holding a *for*, a *step*, a *seq* and an *element* expression was successfully tested for query performance using the testing procedures described in Section 5.1.

5.3 Result Tests

SEQ result test

The input to Pathfinder was a nested sequence (10,(20,30)) and the compiled SQL query was relatively small being able to display the whole SQL code in Figure 5.2. As only simple types were used in the sequence, no access to the database apart from the `loop` Table 5.1(b) was necessary. Hence, only the result schema is given back as shown in Table 5.1(a).

(a) iter	pos	pre	val	(b) loop: iter
----	----	----	----	----
1	1	-	10	1
1	2	-	20	
1	3	-	30	

Table 5.1: The result of the sequence and the loop table

The first tuple in the result table with `pos` as 1 corresponds to the first part of the outside union in lines 4 to 7 of Figure 5.2 being a sequence with the length of 1 containing 10 as a constant. The additional attribute `ord` is set to 1 in line 4.

The second part of the union is again a sequence with (20,30) in lines 10 to 21.

This sequence as a whole holds an `ord` value of 2 in line 16 to ensure that the items "20,30" occur after the item with "10". The inner sequence (20,30) is ordered the same way. The outermost query then gives the positions to each tuple displaying the flattened result in Table 5.1.

```

1 WITH result (iter,pos,pre,val) AS
2 (SELECT e.iter, DENSE_RANK() OVER( PARTITION BY e.iter
3   ORDER BY e.ord,e.pos) ,e.pre,e.val FROM
4   ((SELECT e1.*,1 FROM
5     (SELECT l.iter,1,CAST( NULL AS BIGINT),CAST('10' AS
6       VARCHAR(30)) FROM loop AS l (iter)) AS e1
7     (iter,pos,pre,val))
8   UNION ALL
9     (SELECT e2.*,2 FROM
10      (SELECT e.iter, DENSE_RANK() OVER( PARTITION BY
11        e.iter ORDER BY e.ord,e.pos) ,e.pre,e.val FROM
12        ((SELECT e1.*,1 FROM
13          (SELECT l.iter,1,CAST( NULL AS
14            BIGINT),CAST('20' AS VARCHAR(30)) FROM loop
15            AS l (iter)) AS e1 (iter,pos,pre,val)) UNION
16          ALL(SELECT e2.*,2 FROM
17            (SELECT l.iter,1,CAST( NULL AS
18              BIGINT),CAST('30' AS VARCHAR(30)) FROM
19              loop AS l (iter)) AS e2
20              (iter,pos,pre,val))) AS e
21            (iter,pos,pre,val,ord)) AS e2 (iter,pos,pre,ord)))
22   AS e (iter,pos,pre,val,ord)) SELECT * FROM result ORDER
23   BY iter,pos

```

Figure 5.2: Compiled SQL code of the sequence (10,(20,30))

iter	pos	pre	val
----	----	----	----
1	1	-	1
1	2	-	10
1	3	-	1
1	4	-	20
1	5	-	2
1	6	-	10
1	7	-	2
1	8	-	20

Table 5.2: The result of the nested FLWOR

FOR result test

As stated, SQL code corresponding to nested FLWOR expressions is quite large and difficult to read in one piece. It can span several size A4 pages depending on the depth of nesting and item binding.

Only the query and the result are presented here being similar to the Query 2.1 in Chapter 2, only that no additions `...return $x + $y` to numeric items were implemented. The query below gives back each bound value inside a sequence instead.

```
for $x in (1,2) return
  for $y in (10,20) return
    ($x,$y)
```

This query again uses simple types only accessing `loop` and revealing no encoded XML nodes from any document table.

The Element with document and result tests

With the element test, two results must be checked for the query - the result and the document representation. The element construction also holds a path query, which is why two inference rules are tested together.

```
element test {/descendant::africa}
```

Reviewing the generated XML documents from XMLgen, a descendant node holding `africa` as a property value in the column `prop` of an encoded table should be found. This node is then nested into a new constructed node `test`. The table used for this test was `A.document`.

The result schema, however, gives back only one tuple

iter	pos	pre	val
----	----	----	----
1	0	4899	-

Table 5.3: The element result

The `pre` value of the result corresponds to the newly created root `test` generated according to the *ELEM* inference rule in Section 3.2. If the whole SQL code of the inference rule is projected to the document schema with `SELECT pre, size, level, kind, prop, frag` to give back the whole new document together with the original document `A.document`, then we get the document result in Table 5.4.

pre	size	level	kind	prop	frag
----	----	----	----	-----	----
0	4899	0	0	site	1
...
4898	0	1	1	\012	1
4899	72	0	0	test	2
4900	71	1	0	africa	2
4901	0	2	1	\012	2
4902	68	2	0	item	2
...
4971	0	2	1	\012	2

Table 5.4: The new document frag 2

In the document result, the `pre` value 4899 corresponds to the `pre` value of Table 5.3 and this is our new root node `test`. The nested node `africa` is found being a child of `test` with level 1 and all of its descendant nodes in `pre` order are shown right up to the last `pre` value 4971. To prove if the size value of `test` is correct, the `pre` value 4899 of `test` is subtracted from the last tuple with a `pre` value of 4971, which results in 72. Note that the `pre` values are densely ranked downwards from the last tuple of `A.document` (`frag=1`) being 4898.

Rather unordinary nodes like the last tuple with a node `kind` holding the value 1 are text elements. They hold a value in `prop` being `\012` and are encoded whitespaces. Every whitespace is encoded together with all elements into the relational representation to comply with the W3C specifications. Whitespaces belong to XML documents. If moved or removed, they are regarded as a different document. So when an XML document is relationally encoded it must be guaranteed

that with the re-transformation back into XML text, the document looks exactly like it was before.

5.4 Time Measurements

The query to be tested regarding performance time was first tested if correct and then run on all seven documents:

```
for $i in /descendant::africa/child::item
  return ('african-items',element african-item{$i})
```

Query 5.5: Tested FLWOR query

A test run on B.document gave back a sequence with four tuples:

iter	pos	pre	val
----	----	----	-----
1	1	-	african-items
1	2	23951	-
1	3	-	african-items
1	4	24024	-

Table 5.6: Result from B.document

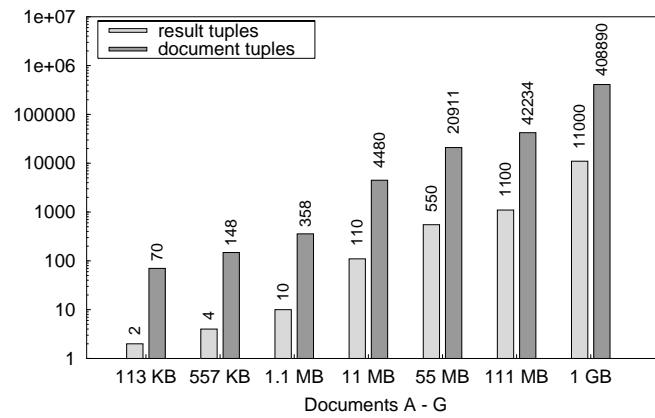
The tuples with pos=1 and 3 are the simple types african-items. The items with pos=2 and 4 are the african-item elements which can be referenced in the document result in Table 5.7.

pre	size	level	kind	prop	frag
23951	72	0	0	african-item	2
23952	71	1	0	item	2
...
24023	0	2	1	\012	2
23951	74	0	0	african-item	3
23952	73	1	0	item	3
...
24098	0	2	1	\012	3

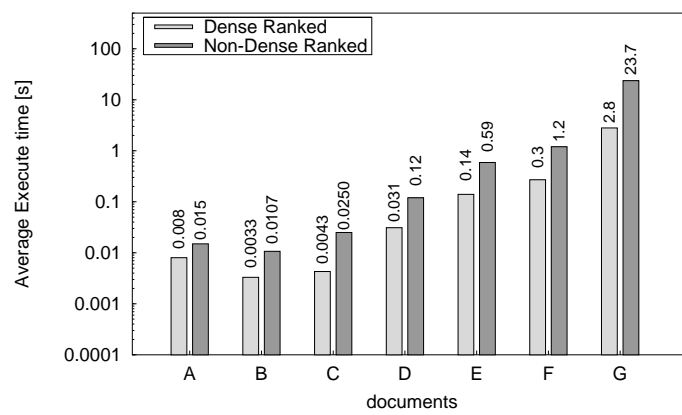
Table 5.7: New document fragments from B . document

The successful query tests show promising performance times in Figure 5.3 on all document sizes. The result schema is used to give back results and intermediate results and the document schema returns all queried document tuples in order to be transformed back into XML text. Figure 5.3 (a) shows the amount of tuples returned to give an impression of the document sizes which have an effect on the query performance times.

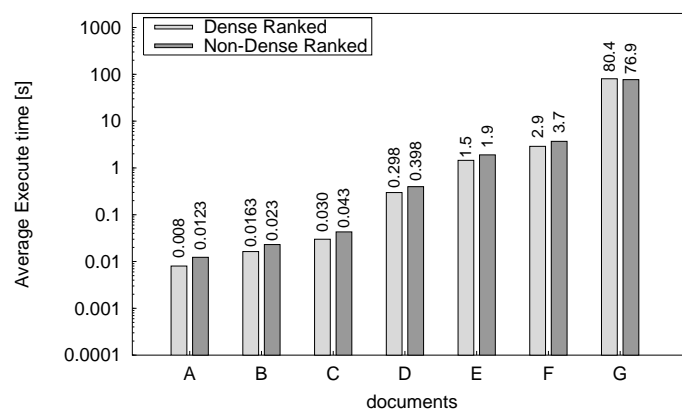
Dense ranking has two advantages: the first is the avoidance of integer overflow, the second is that the queries show far better performance times on all tested document sizes with the result schema (see Figure 5.3 (b)). These queries can be almost up to ten times faster than non-dense-ranked queries as shown in Figure 5.3 (b) for document G. The query times with the document schema also prove to be shorter using dense ranking, apart from the query for the largest document G (see Figure 5.3 (c)).



(a) Tuples returned



(b) Result time measurements



(c) Document time measurements

Figure 5.3: XML documents tested by Query 5.5

Chapter 6

Discussion

6.1 Relational Encoding

The relational backend of the Pathfinder project certainly presents a promising basis for encoded XML documents on RDBMS. The one to one storage scheme - storing one whole XML document into a single database table - defeats the problem of having to access multiple tables of encoded nodes. The queries generated by the implementation only access this table together with the loop table holding a singleton `iter` value.

The concept of the relational result schema, with both simple- and element types in a sparse form, supports one central part of the XQuery requirements. Furthermore, this schema can express the encoding of sequences, a fundamental data structure of XQuery, as well as enable nesting in `for`-loops by exposing each single iteration of a sequence.

This result schema also adapts very well to the document encoding developed by Grust et al. ([Gru02] and [GvKT03]) by only holding the `pre` value of a result node in a result schema which can be referenced in a document.

Of course, drawbacks of the document encoding must be mentioned concerning inserts and deletions of nodes. Inserting or deleting a node from a persistent document table would call for recalculations of attributes holding structural information such as `pre`, `size` and `level`. The recalculation load can vary, depending on the location of the inserted or deleted node as well as the size of the persistent table as a whole.

It would not be correct to say that the document schema fully encodes XML. Attributes have not been dealt with in the implementation and do not occur in the document encoding as such. They are not regarded to be children of an element but have a defined position in the document order after their owning element. As elements can hold several attributes and as attributes are regarded to be of a dif-

ferent nature, a separate storage of attributes is also provided by the relational backend of Pathfinder. For every document table a further attribute table is held in the database containing all attributes appearing in the document. Each attribute tuple holds its own attribute id, the element owner as a foreign key to the document, an attribute name and an attribute value (see Table 6.1(b)).

(a)	pre	size	level	kind	prop	frag
	-----	-----	-----	-----	-----	-----
	5	3	2	0	item	1

(b)	att_id	owner	name	value		
	-----	-----	-----	-----		
	1	5	price	50		
		

Table 6.1: Element with attribute

So further development towards attribute support is desirable in order to extend relational XML encoding. Attributes should appear in the result schema but be distinguishable from other node types. The inclusion of attributes means that access to more than only the document and loop table but also to the corresponding attribute table over join operations would be necessary.

6.2 XQuery to SQL Refinements

As the aims of this thesis were to make a way through to finally be able to perform SQL queries from compiled XQuery core expressions, further implementation and optimisation hooks are certainly a matter of interest.

As mentioned in the *STEP* inference rule of Section 3.2, the problem of duplicate removal using the Staircase Join approach by [GvKT03] as well as the Path Stack algorithm developed by [NBS02] may become useful.

In the performance testing phase certain unfavourable path queries caused extremely bad performance results and can possibly be defeated by form of query rewriting as shown by [OMFB02]

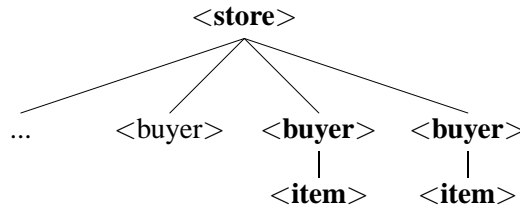
- The approach to minimise the duplicate problem by applying the Staircase algorithm causes necessary changes to the *STEP* rule of the implementation. The Staircase algorithm accesses the path query in a flattened manner $e/\alpha_1::n_1/\alpha_2::n_2/.../\alpha_k::n_k$

other than the step bundling of our implementation in the form of

$(..((e/\alpha_1::n_1)/\alpha_2::n_2)/..)/\alpha_k::n_k$

The action code of the *twig* rules for *STEP* would first have to gather all path steps to then provide the full path to Staircase Join instead of emitting SQL node constructors for each step. As Staircase Join supports all XPath axes the full implementation of all axes in the *twig* rules would be desirable to enable further performance testing.

- The Path Stack algorithm can also take advantage of flattened path steps by pre-scanning the document and pushing the `pre` indexes of context nodes onto stacks that correspond to the full path expression.



Each context node n_1 to n_k owns a separate stack $S[1]$ to $S[k]$ each holding $(n^1, n^2 \dots)$ valid node indexes.

At the end of a document scan all path query matches

$S[1](n^1 \dots n^j), S[2](n^1 \dots n^m) \dots S[k](n^j \dots n^n)$

are available and can be given back by backtracking the stacks and giving back all node sequences representing valid paths

$(n_1^j, n_2^m \dots n_k^n), (n_1^{j-1}, n_2^m \dots n_k^n), \dots (n_1^1, n_2^1 \dots n_k^1)$. The Path Stack algorithm, however, only supports descendant and child axes but can become useful when pre-scanning a document and forcing a caching of all the valid paths when frequent access to a given path query is necessary. A more detailed description of the Path Stack algorithm is given by Bruno et al. [NBS02] together with further information on optimisations using a technique to support whole XML tree matches.

- As a last optimisation hook, query rewriting by Olteanu et al. [OMFB02] may be useful by performing rewriting on reverse axes such as ancestor, preceding and parent.

Path queries, as for example: `/descendant::n/parent::m`, can be rewritten to `descendant-or-self::m[child::n]`. The decision on which rewrite rule should be applied is based on heuristics and cost estimations.

Performing such rewrite rules on reverse axes can be beneficial performing only forward sequential index scans on encoded documents.

Each of the three mentioned optimisation hooks are not necessarily beneficial in all cases and leave room for discussion on further developments towards relational back-ends.

6.3 Future Development on RDBMS

The major problem of compiling XQuery into SQL code as described in the performance measurements is the size of the output SQL code. Here, necessary downsizing of SQL code is desirable. Although the built-in functions of the SQL:1999 standard such as `COALESCE()` and OLAP's `DENSE_RANK()` functionality have been proved to be very useful to the implementation, they do not contribute to minimising SQL code size. Regarding RDBMS development, built in functions supporting XQuery `for-loop` and `element` nesting are necessary in order to shift large parts of the query workload to the RDBMS.

Chapter 7

Attachment

SQL Test Queries:

All the tested SQL queries can be taken from the attached CD-ROM.

Bibliography

- [AGT89] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. In *ACM Transactions on Programming Languages and Systems, Vol. 11, No. 4*, pages 491–516, October 1989.
- [CDFK03] D. Chamberlin, D. Draper, M. Fernández, and M. Kay. *XQuery from the Experts*. Boston: Addison-Wesley, 2003.
- [DTCO03] D. DeHaan, D. Toman, M. P. Consens, and M.T. Özsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *Proceedings of the 22nd International ACM SIGMOD Conference on Management of Data, San Diego, CA*, pages 623–634, June 2003.
- [FK99] D. Florescu and D. Kossman. Storing and Querying XML data using an RDBMS. In *Data Engineering Bulletin 22(3)*, pages 1–15, 1999.
- [Gru02] T. Grust. Accelerating XPath Location Steps. In *Proceedings of the 21st International ACM SIGMOD Conference on Management of Data, Madison, Wisconsin, USA*, pages 109–120, June 2002.
- [GST04] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB 2004), Toronto, Canada*, pages 1–12, August / September 2004.
- [GvKT03] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach A Relational DBMS to Watch its Axis Steps. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB 2003), Berlin, Germany*, September 2003.
- [KKN03] R. Krishnamurthy, R. Kaushik, and J. Naughton. XML-to-SQL Query Translation Literature: The State of the Art and Open Problems, Berlin, Germany. In *Proceedings of the 1st International XML Database Symposium (XSym)*, pages 1–18, September 2003.

-
- [NBS02] N. Koudas, N. Bruno and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the 21st International ACM SIGMOD Conference on Management of Data, Madison, Wisconsin, USA, June 2002*.
- [OMFB02] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *EDBT Workshop on XML-Based Data Management, Prague, Czech Republic, 2002*.